Automatic Test Case Generation of Confromance Testing for Communication Protocols Specified in Extended Models

By

Chang-Jia Wang, Ph.D.

The Ohio State University, 1994

Ming T. Liu, Adviser

A *communication protocol*, which is a set of rules governing the operation of a computer network, is essential for supporting the network's reliability. The written statement of these rules is called a *protocol specification*, which cannot function unless it is *implemented*. However, due to human errors or incompatibilities between the protocol and its lower layer systems, a protocol implementation may not behave as specified. Thus, *conformance testing* is used to ensure the consistency between a protocol specification and its implementation. An important area of conformance testing is *test case generation (TCG)*, which automatically generates a set of *test cases* to discover any discrepancy between a protocol specification and its implementation.

Test case generation has been studied in the literature for protocols specified by the *Finite State Machines (FSMs)* model. However, many practical, real-world

protocols are hard or infeasible to be specified by the FSM model because the model contains no memory. As the result, more sophisticated models with memory, called *extended models*, are used in practice. Although extended models can dramatically simplify a protocol specification, both the control and data flows of the specification need to be considered in order to generate a test case. Unfortunately, because of the difficulty in tracing the data flow, generating a test case for extended models remains a challenging problem, and little research has been done on this area.

In this dissertation, the test case generation problem for extended models is introduced. Because of its simplicity and generality, *EFSM (Extended Finite State Machine)* is used as the model in this dissertation. It is found that *axiomatic semantics* is a suitable technique for monitoring the data flow of an EFSM. Using axiomatic semantics as a tool, several methods for EFSM test case generation are proposed. The first method, called *ATG* (Axiomatic Test case Generation), generates test cases for the *single-transition fault*. Based on the same technique, the second method, called *OAT* (Optimized ATG), uses more complicated axioms to derive shorter test cases.

Next, a *behavior model* is proposed to describe the dynamic behavior of an EFSM. Using the model, *TGF* (Test case Generation for given Fault models) is used to generate a test case based on a given fault model. This method is also extended to generating test cases for an OSI standard specification language, Estelle. Finally, it is found that TGF can be transformed into a protocol validation problem. Therefore, many existing protocol validation tools can be used to generate test cases. As a result, a test case generator is implemented using an existing protocol validation tool.

Automatic Test Case Generation of Confromance Testing for Communication Protocols Specified in Extended Models

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Chang-Jia Wang, B.S., M.S.

* * * * *

The Ohio State University

1994

Dissertation Committee:

Ming T. Liu

Phillip E. Krueger

Ten-Hwang Lai

Approved by

____

Adviser
Department of Computer
and Information Science

To my parents and my wife

Acknowledgements

I dedicate my deepest appreciation to my father, who taught me how to write my first computer program. I still remember it was a BASIC program that summed up numbers from one through five. The joy and excitement after struggling in debugging caught me into this field. At the time when only handful of companies could afford to have a computer, being able to play one was an experience that other kids could only dream of. Now, I am following his foot steps and hoping someday, I will become a great computer scientist as he is today.

My thanks are also due to my mother, who took care of everything, especially at the time when my father was studying abroad and at the time when I was so ill that doctors predicted I had only six months to live. All these years I have been brought up without knowing what trouble is. Only after I came abroad then I realized that life has so many things to worry about.

I also appreciate Professor Ming T. Liu, who had belief in me when others questioned my ability. Without his effort, I could not possibly enter this program. He not only advised my academic work, but also taught me things that cannot be learned from anywhere else. I thank him for his patience when he corrected careless errors in my papers and for his forgiveness when I keep goofing up and forgetting important things.

I am grateful to Professor Phillip Krueger and Professor Ten-Hwang Lai for serving as members of my reading committee, and to the members of Advanced Communication Systems (ACS), especially Wen-Shyen Chen, Yo-Wei Yao, Sarah Yu and Lian Seng Koh, for countless advice that flourished into the ideas presented in this dissertation. Li-Fen Wu, Yu-Chi Tseng and Xuefong Dong have always been my moral support and I owe them a lot.

My wife Meei-Ling was the driving force behind those published papers. From the day we first met, she has started encouraging me to write and publish. Without her, I would not have started this dissertation until next year, or even later.

The work reported in this dissertation has been supported by U.S. Army Research Office under contracts No. DAAL03-91-G-0093 and No. DAAL03-92-G-0184. The views, opinions, and/or findings contained in this dissertation are those of the author and should not be construed as an official Department of the Army position, policy or decision.

VITA

Nov. 3, 1962	Born: Kaohsiung, Taiwan, Republic of China
Jun., 1985	B.S., Department of Computer Engineering National Chiao Tung University, Hsinchu, Taiwan.
Sep., 1988 – May 1989	Graduate Research Assistant, Department of Computer Engineering University of Southern California, Los Angeles, California.
Jun., 1989	M.S., Department of Computer Engineering University of Southern California, Los Angeles, California.
Jan., 1990 – present	Graduate Research Associate, Department of Computer and Information Science The Ohio State University, Columbus, Ohio.

Publications

Research Publications

C.-J. Wang and M. T. Liu "Axiomatic Test Sequence Generation for Extended Finite State Machines". *Proceedings of 12th International Conference on Distributed Computing Systems*, pp. 252–259, June 1992

C.-J. Wang and M. T. Liu "A Test Suite Generation Method for Extended Finite State Machines Using Axiomatic Semantics Approach". *IFIP Transactions on Protocol Specification, Testing, and Verification, XII*, pp. 29–43, North-Holland, 1992

C.-J. Wang and M. T. Liu "Generating Test Cases for EFSM with Given Fault Models". *Proceedings of IEEE INFOCOM '93*, pp. 774–781, April 1993

C.-J. Wang, P. Krueger, and M. T. Liu "Intelligent Job Selection for Distributed Scheduling". *Proceedings of 13th International Conference on Distributed Computing Systems*, pp. 517–524, May 1993

C.-J. Wang, and M. T. Liu "Automatic Test Case Generation for Estelle". *Proceedings of 1993 International Conference on Network Protocols*, pp. 774–781, October 1993

C.-J. Wang, L.-S. Koh, C.-H. Wu, and M. T. Liu "A Multimedia Synchronization Protocol for ATM Networks". *Proceedings of 14th International Conference on Distrubuted Computing Systems*, June 1994

Fields of Study

Major Field: Computer and Information Science

Studies in:

Computer Networks and Architectures	Prof. Ming T. Liu
Operating Systems	Prof. Phillip E. Krueger
Distributed Algorithms	Prof. Ten-Hwang Lai

Table of Contents

List of Tables

List of Figures

CHAPTER I

Introduction

1.1 Background and Motivation

Recently, the rapid development of *computer networks* has created a cooperative computing environment, in which computers around the world can share resources, information, and workload. As the result, people can retrieve information from data banks located thousands miles away, held a meeting or work on a project with other persons in other continents, use supercomputers that they cannot afford, and watch TV programs or read up-to-the-minute news specially edited for their own tastes. Being the backbone of these services, the underlining network must provide adequate performance while maintaining its reliability. Therefore, *communication protocols* are designed to equip the network with such capabilities.

A *communication protocol* (or simply a *protocol*) is a set of rules governing the operation of a computer network. A written statement that describes these rules is called a *protocol specification* (or a *specification* in short), which cannot function unless it is *implemented.* Ideally, a *protocol implementation* (or an *implementation*) is supposed to behave consistently with its specification. Unfortunately, because of the incompatibility between a protocol and its underlying system, or human errors such as misinterpretation of the specification or mistakes occurred in program design, the behavior of a protocol implementation may not be what had been expected. Therefore, a research area called *protocol conformance testing* is emerged to ensure the consistency between a protocol specification and its implementation.

One of the most important areas of conformance testing is *test case generation (TCG)*. Given a protocol specification, the goal of TCG is to construct a set of *test cases*, each of which contains a sequence of inputs and a sequence of expected outputs. An unexpected output sequence may occur when the input sequence is applied to an incorrect protocol implementation. Since a protocol usually includes several programs running in parallel at different machines, it is very difficult to construct test cases manually to reveal implementation faults. Thus, being able to generate test cases automatically is highly desirable.

One protocol can be specified by many different *specification models*. Most specification models can be classified into the following three classes.

1. *State-transition models*: including *Finite State Machines (FSMs)* and *Petri-Nets*,

2. *Programming language models*: such as *CSP*, *Promela*, *LOTOS* and other generally used programming languages, and

3. *Hybrid models*: such as *Extended Finite State Machines (EFSMs)* and Estelle.

Since a TCG method generates test cases based on a protocol specification, it is usually designed for a particular specification model. So far, most of the TCG methods proposed in the literature are based on FSMs. An FSM is usually described as a directed graph. The nodes of the graph represent the *states* and the directed edges are called *transitions.* A transition is labeled by an input symbol and an output symbol. A protocol specification moves from state A to state B if the input symbol of the transition pointing from A to B is received. Upon moving into state B, the FSM also produces an output according to the output symbol shown on the transition [20, 39, 54]. An FSM is simple, easy to specify, and easy to understand; however, it contains no memory and is hard to specify many complicated protocols. Even one can work around such problems (by using different states to represent different values in the memory), the resulting FSM will be so complicated that it not only loses its advantage of comprehensibility, but also becomes too difficult for a test case generator to generate feasible test cases (see Section 4.1).

Instead of using FSM, most real-world protocol specifications are written either in programming languages (e.g., LOTOS) or in models that are the hybrid of programming languages and FSM (e.g., EFSM and Estelle)[1]. These specification models (called *extended models* in this dissertation with respect to the basic FSM model) usually contain memory, so a specification can be dramatically simplified. However, a TCG method for an extended model needs to consider not only the control flow, but also the data aspects of the protocols (Chapter IV). Unfortunately, since it is very difficult to track the data flow, TCG for extended models remains a challenging problem. As a result, not much research has been done so far. The purpose of this

[1]Some specifications are written in plain English. TCG for informal descriptions is extremely difficult, if possible at all, and is not considered in this dissertation.

dissertation is to discuss the issues in the TCG problem for extended models and to propose feasible solutions to the problem.

1.2 Previous Work

From the handful TCG methods considering the data aspect, one can discover two streams of research. One uses LOTOS [4, 27] and the other uses EFSM or Estelle [7, 13, 26] as their protocol specification models. The research based on LOTOS usually uses mathematical notations to describe TCG problems, but does not create any applicable test cases. Therefore, it is more like a theoretical description for the TCG problem than proposing practical methods for the problem (Section 4.3). As to the TCG methods for EFSM or Estelle, most of them are based on one of the following two approaches. One is to convert an EFSM to an FSM and generate test cases for the FSM using existing FSM TCG methods. However, the generated FSM either removes some functionality from the original EFSM, or becomes too complicated for an FSM TCG method to generate feasible test cases. The other approach is to generate data flow graphs to describe the interactive relationship between data items. Unfortunately, the papers published for this approach usually concentrate on the generation of the data flow graphs. Little has been addressed on how to generate applicable test cases (Section 4.2).

It is quite disappointing that after both LOTOS and Estelle have been adopted as OSI standard specification languages for communication protocols for some time, there are still no TCG methods available to generate applicable test cases for them. Thus, this dissertation is dedicated to finding a way to generate actual test cases for extended models. Since this is quite a pioneering work in the area, without losing any specification power, a simplified EFSM model is used (Section 2.3).

1.3 Proposed Approach and Contribution

To create test cases for extended models, one must have a way to trace the changes and dependencies of the data items. A technique used in program proving (or program verification), called *axiomatic semantics* [45], is found to be a useful tool for serving such a purpose. Axiomatic semantics provides a set of *axioms* that describes the general rules of how the status of a program, called *assertions*, is changed before and after a statement (see Section 2.5 for detail). Depending on how the axioms are written, one can monitor different properties of a program and use the properties for different purposes. It is found that one can design the axiom in such a way that after a program is evaluated, the resulting assertion contains a test case. Base on this idea, two TCG methods for EFSM models that utilize the axiomatic semantics approach are proposed (Chapters V and VI) [57, 58].

Almost every TCG method appearing in the literature generates test cases to detect a specific type of error. The error is called a *fault model* [55]. Since a test method is designed for a fixed fault model, the ability of detecting errors in the implementation is limited. For example, suppose any existence of a critical fault in a protocol implementation significantly affects the correctness of the protocol. If unfortunately, this fault is not included in the fault model of the TCG method being used, one will be forced to use other TCG methods or to give up testing the fault. Besides, most TCG methods with fixed fault models do not explicitly indicate their fault models. When a protocol implementation passes a test case (generates expected output with respect to the input sequence of the test case), one still cannot be sure of the absence of the critical fault. Hence, a TCG method that generates a test case for a given fault model is proposed (Chapter VII) [60]. This method has also been extended to generating test cases for the ISO standard specification language, Estelle (Chapter VIII) [59].

Finally, it is found that a TCG method based on fault models can be transformed into a *protocol validation* problem. Protocol validation is used to determine whether a protocol specification is "correct," e.g., free of deadlocks. This area has been studied intensively for years, and many validation tools are available. By transforming a TCG problem into a validation problem, a protocol validation tool can be used to generate test cases (Chapter IX). As a result, a test case generator is implemented by using SPIN [21], a protocol validation tool, as its backbone.

1.4 Organization and Overview of the Dissertation

The models used in this dissertation are defined in Chapter II. The ISO framework for conformance testing is introduced in Section 2.1, which defines the properties, requirements, as well as terminology used by ISO for conformance testing. Two most generalized models used in this dissertation, FSM and EFSM are introduced in Sections 2.2 and 2.3, respectively. Finally, axiomatic semantics, which is the technique used in the proposed TCG methods, is described in Section 2.5.

Previous work for the TCG problem is briefly discussed in both Chapters III and IV. In Chapter III, TCG methods for the FSM model, as well as some of their test case optimization methods are introduced. In Chapter IV, TCG methods for EFSM, Estelle, and LOTOS are described, and the pros and cons of these methods are discussed.

The contributions of this dissertation are described in Chapters V through IX. In Chapter V, *ATG* (*Axiomatic Test case Generation*), a TCG method based on the axiomatic semantics technique, is proposed. The method generates a test case for each individual transition in an EFSM. The axioms used in ATG are relatively simple, and therefore are used as an introduction to axiomatic semantics test case generation. Since test cases are generated for each transition separately, the test cases generated are mostly redundant. This problem is resolved in the *OAT (Optimized ATG)* method, which is described in Chapter VI. OAT uses more complicated axioms and considers several transitions at once to minimize the redundancy.

As mentioned in Section 1.3, all the methods discussed in Chapters III to VI are either concerning with only a fixed fault model, or not considering any fault model at all. These methods are lack of the ability to provide users with the confidence of the protocol implementation's correctness. In Chapter VII, *TGF (Test case Generator with given Fault models)*, which generates test case for given fault models is introduced. The method is extended, and applied to OSI standard specification language, Estelle, in Chapter VIII.

In Chapter IX, TCG problem based on fault model is transformed into a protocol validation problem (*ValTest, Validation based Testing*). It is shown that finding a test case can be reduced to finding an deadlock in a concurrent system. As a result, an existing protocol validation tool can be used to generate test cases with virtually no extra cost.

Finally, in Chapter X, a conclusion is given. The chapter summarizes this dissertation and states some future research issues for TCG of extended models.

CHAPTER II

Models for Conformance Testing

2.1 OSI Framework for Conformance Testing

2.1.1 Definition of Conformance

Nothing can be discussed further in this dissertation without a definition of conformance. According to [48], conformance is the consistency "of implementations and real systems to those OSI standards which specify applicable requirements." In general, conformance is the consistency between the protocol implementations and their specifications.

There are two types of requirements for conformance, namely *static conformance requirements* and *dynamic conformance requirements.* The static conformance requirements states the *minimum* capability that an implementation must provides, while dynamic conformance requirements restricts the *maximum* capacity allowed for an protocol implementation. In other words, static requirements state things that an implementation should have, and dynamic requirements state things that must not happen. Ensuring an implementation satisfies the static requirements is called *weak conformance* and ensuring it matches the dynamic requirement is called *strong conformance* [52].

2.1.2 Test Methods

The OSI protocol standards [25] defines the behavior of a protocol entity in term of *Protocol Data Units (PDUs)* and *Abstract Service Primitives (ASPs)*. In the OSI layered model, the entities on the N-th layer communicate with each other through the PDUs of the layer, called N-PDUs. However, there may not be a direct link between the entities on the N-th layer, so the PDUs are transformed into $(N-1)$-ASPs and carried by the $(N-1)$-PDUs of the $(N-1)$-th layer (Figure 1). That is, the $(N-1)$-th layer provides communication services for its upper layer N.

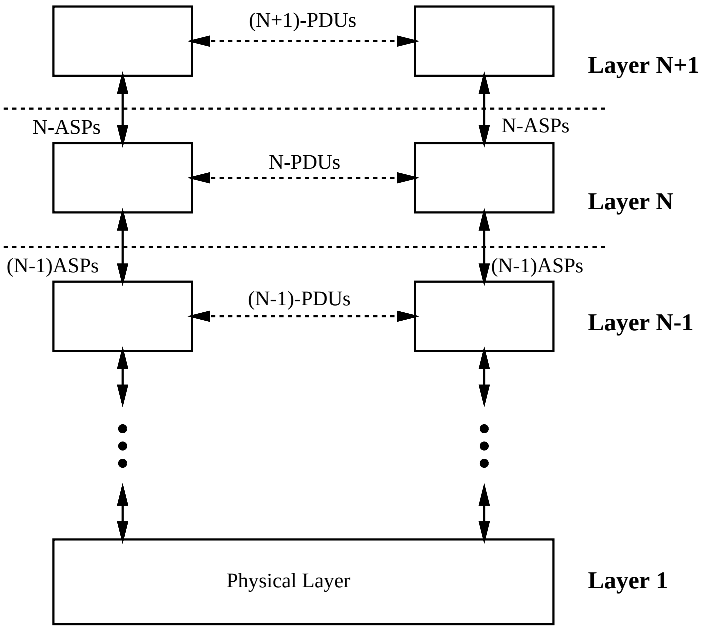


Figure 1: The OSI layered model for communication protocols

The OSI framework of test methods is also described in term of PDUs and ASPs [28, 29, 30, 31, 32]. The protocol entity being tested is called an *Implementation Under*

Test (IUT). When an IUT is tested, it is embedded in a test bed, called *tester*, which simulates the environment in which the IUT would be operated normally (Figure 2). The tester can either be a program or a human being who types in inputs and observes the outcomes.

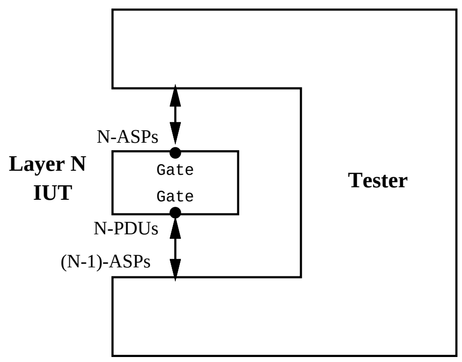


Figure 2: IUT and its tester

The IUT is considered a black box; i.e., no detailed information about the IUT's structure is available. The only communication between the IUT and the tester is through the *gates*. A gate is an I/O port of an IUT, through which PDUs and ASPs can be exchanged.

2.1.3 Test Suites

The discrepancy between a protocol specification and its implementation are called the *faults* of the implementation. A *test suite* is a set of statements describing how to control and observe the IUT through the gates, so that the faults in the IUT will likely be detected.

In OSI test framework, a test suite can be divided into *test groups*, a test group is further divided into *test cases*, a test case is divided into *test steps*, and a test steps is composed of a sequence of *test events.* A test event is a single input or output action expected to be performed by the IUT. This is similar to a book being divided into chapters, and a chapter being divided into sections. In this dissertation, this hierarchy is simplified into three levels: test suite, test case, and test events. Each test case is in charge of detecting a certain fault in the IUT, and is composed of a sequence of test events. The test event sequence states the appropriate inputs that should be fed into an IUT, and the expected outputs that can be observed from the IUT. The faults that a test case tries to detect is called the *test purposes* of the test case.

2.1.4 Test Merit

The quality of a test case can be measured either by its *length* or by its *fault coverage.* The length of a test case is the number of test events contained in the test case. The longer the test case, the more time and resource are needed to perform the test. Therefore, a shorter test is more desirable. The fault coverage of a test case is the number of faults a test case can detect. The larger the fault coverage, the easier a test case can point out an erroneous implementation. Therefore, a larger fault coverage is more desirable.

A TCG method needs to generate test cases as short as possible while extending their fault coverage as wide as possible. However, these two quality often conflict.

Trade-offs are usually needed for a TCG method to determine a suitable test case [56].

2.1.5 Test Procedure

A test case is basically composed of an input event sequence and an output event sequence. The input and output event sequences are carefully designed such that, when the input is fed into the IUT, an unexpected output will occur if the IUT contains the fault that the test case trying to detect. This procedure is illustrated in Figure 3.

A problem remained to be solved is how to obtain the test cases from the specification. Currently, the test cases are often designed *ad hoc* for individual protocol. Therefore, constructing a test case requires tremendous work and, as well as the protocol implementation that the test case tries to test, is prone to error. Hence, it is desirable to automate the test case generation process (the shaded area in Figure 3), which is the main purpose of this dissertation.

2.2 Finite State Machines

A Finite State Machine (FSM) [54, 20, 39] is usually described by a directed graph, called *state-transition* diagram. The nodes in the diagram represent the *states* and the edges indicate the *transitions*. The state from which a transition starts is called the *head state* of the transition, and the state in which a transition ends is called the *tail state*. A transition is an *outgoing transition* of its head state and is an *incoming transition* of its tail state. An FSM moves from one state to another by traversing

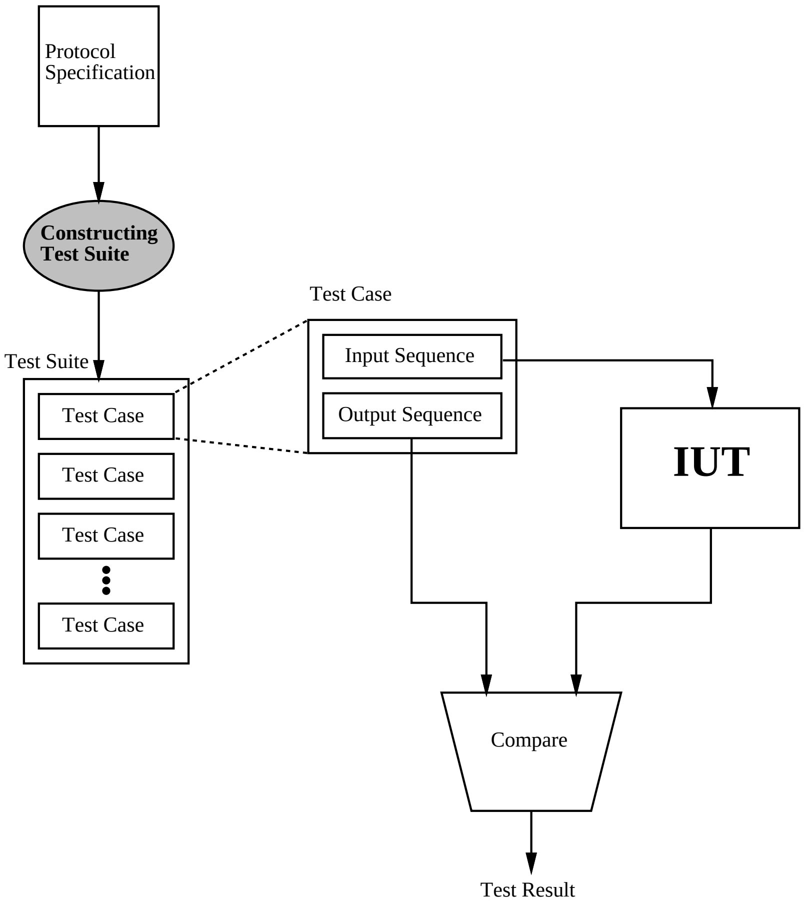


Figure 3: Test procedure

through the transitions between the states. There is one and only one state, called the *initial state*, from which an FSM starts. In a state-transition diagram, the initial state is pointed by a transition without a head state. It is assumed that there is a reset signal through which an FSM can be set to the initial state, regardless in which state the FSM originally locates.

A sequence of transitions is called a *path*. The head state of the first transition in a path and the tail state of the last transition in the path are called the head and tail states of the path, respectively. A path with head state and tail state being the same is called a *tour*. The *length* of a path is defined as the number of transitions in the path.

Every transition in the diagram is associated with a label, which indicates the input and output actions that take place while the transition being traversed. The input action is written on the left hand side of a slash ("/"), and the output action is written on the right. In this dissertation, the input and output actions are specified by CSP-like notations, $G?m$ and $G!m$, respectively, where G is a gate and m is the message being input or output through gate G. An example of an FSM is shown in Figure 4, which specifies the sender side of the *Alternating Bit Protocol (ABP)*.

In Figure 4, the FSM communicates with its upper layer protocol at the local site and its pear receiver at the remote site through gate U and gate R, respectively. Starting from the initial state a, the FSM enters state b when its upper layer sends it a message M through gate U (transition 1), demanding the sender send a data to its pear receiver. The FSM then sends a data $D0$ through gate R after receiving the

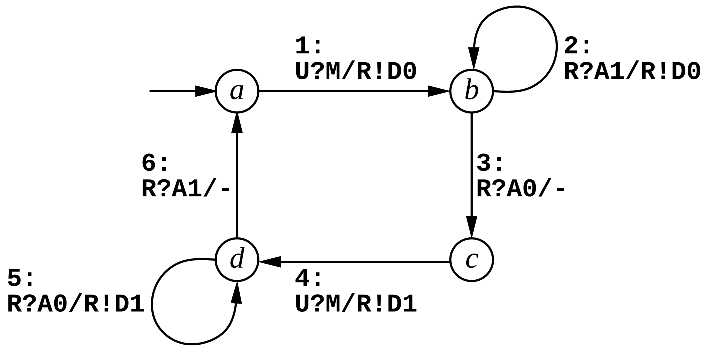


Figure 4: The sender of ABP specified in FSM

message M. The "0" in the data item marks sequence number of the data. When the receiver replies acknowledgment $A0$ (transition 3), which has the same sequence number as the data just sent, the FSM enters state c, realizing that the data has been received correctly. The dash ("–") in transition 3 indicates no output being produced. On the other hand, if an $A1$ is received instead (transition 2), errors must have occurred. The FSM retransmits $D0$ again and goes back to state b. Transitions 4, 5, and 6 work similar to transitions 1, 2, and 3, respectively, except that the roles of sequence numbers 0 and 1 are reversed.

An FSM can be formally defined as follows.

Definition 2.2.1 *An FSM is a quintuple, $\langle Q, s, E, \Sigma, \delta \rangle$, where*

1. *Q is a set of states,*

2. *$s \in Q$ is the initial state,*

3. *$E \subseteq Q$ is a set of final states,*

4. *Σ is a set of symbols (or labels), which denote the I/O actions for the transitions, and*

5. *$\delta : (\Sigma \times Q) \mapsto (\Sigma \times Q)$ is a transition function.*

Note that δ is a function, which means the FSM is *deterministic*. An FSM is deterministic if at any given state and provided with a given input, the FSM can only traverse through exactly one transition. Otherwise, it is *nondeterministic*. Most of the FSM TCG methods mentioned in this dissertation are based on deterministic FSM model. A formal definition of the FSM in Figure 4 is shown in Figure 5.

1. $Q = \{a, b, c, d, e, f\}$
2. $s = a$
3. $E = \emptyset$
4. $\Sigma = \{U?M, R!D0, R!D1, R?A0, R?A1, -\}$
5. $\delta(U?M, a) = (R!D0, b), \quad \delta(R?A1, b) = (R!D0, b), \quad \delta(R?A0, b) = (-, c)$
   $\delta(U?M, c) = (R!D1, d), \quad \delta(R?A0, d) = (R!D1, d), \quad \delta(R?A1, d) = (-, a)$

Figure 5: Formal definition of the FSM in Figure 4

2.3 Extended Finite State Machine

An Extended Finite State Machine (EFSM) is an FSM with memory called *variables*. It behaves similarly to an FSM except that an EFSM associates a transition with a statement (or an *action*) rather than labeling a transition with a pair of input and output symbols. Without losing the generality, the statement of a transition can be one of the following.

1. An input action $G?x$, where G is a gate and x is a variable. When the transition with an input action is traversed, an input is received from gate G and assigned to variable x.

2. An output action $G!e$, where G is a gate and e is an expression. The result of expression e is sent to gate G if the transition has the output action.

3. An assignment action $x := e$, where x is a variable and e is an expression. The assignment action calculates the result of expression e and assigns the result to variable x.

4. A conditional action $\mathcal{C}$, where $\mathcal{C}$ is a boolean expression. A transition with a conditional action cannot be traversed unless the boolean expression is satisfied.

The input and output actions are called *external actions*, which can be observed from the outside. On the contrary, the assignment and the conditional actions are *internal actions* since they are actions taken internally without the notice of the outside world. The instance that an action is performed is called an *event*. That is, when an external action is executed, an *external event* occurs. Similarly, an *internal event* occurs when an internal action takes place.

A transition with an input, output, assignment or conditional action, is called an input, output, assignment or conditional transition, respectively. Similarly, an internal (external) transition is a transition with an internal (external) action.

An example of an EFSM specification for the sender of ABP is shown in Figure 6. In the figure, the sender communicates with its user (upper layer protocols) at the local site and its peer receiver at a remote site through gates U and R, respectively. Starting from the initial state a, the sender of the ABP resets variable B to 0 (transition 1) and reaches state b. When it receives a message from the user (transition 2), it stores the message in variable M and goes to state c. Through transition 3, the sender sends message M with variable B as the message sequence number and reaches state d. When the sender receives the acknowledgment (transition 4), it stores the sequence number of the acknowledgment in variable A and goes to state e. If the sequence number of the acknowledgment matches the sequence number of the message (transition 6), the sender goes to state f. Otherwise (transition 5), it goes back to state c and retransmits the original message M. At state f, the acknowledgment indicates that the message sent to the receiver was correctly received at the remote site. Then, the sender alters the current sequence number of the message B (transition 7), goes back to state b and gets itself ready to transmit the next message for its user.

Unlike an FSM, a path in an EFSM may not always be executable. For example, a path containing transitions $\langle 1, 2, 3, 4, 5 \rangle$ in Figure 6 is not a possible execution sequence if a one is input in transition 4. A path is *legal* if it is a feasible execution sequence. In this dissertation, unless mentioned explicitly, a path usually means a legal path.

An EFSM can be defined formally as follows.

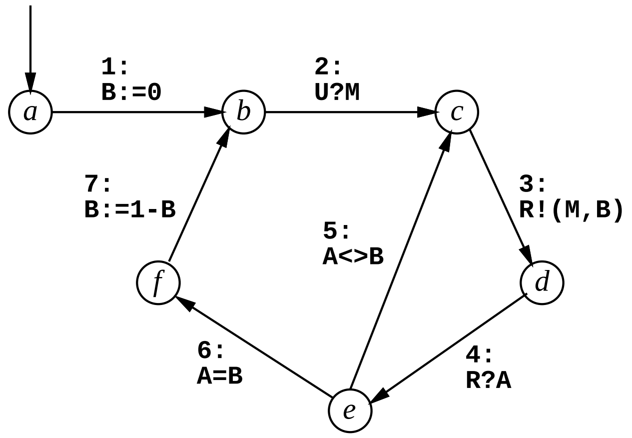


Figure 6: An EFSM specification for ABP

Definition 2.3.1 *An EFSM is an octuple,* $\langle Q, s, E, V, \tau, K, R, \Delta \rangle$*, where*

1. Q *is a set of states,*

2. $s \in Q$ *is the initial state,*

3. $E \subseteq Q$ *is a set of final states,*

4. V *is a set of variables,*

5. K *is a finite or infinite set of values,*

6. $\tau \in V$ *is an imaginary variable that records the current external event,*

7. $R = \{\rho \mid \rho : V \mapsto K\}$ *is a set of status, and*

8. $\Delta \subseteq (Q \times R) \times (Q \times R)$ *is a transition relation.*

Note that Δ is a relation; therefore, the EFSM can be nondeterministic. The status of an EFSM, R, is a mapping from the variable set V to the value space

K. That is, a status indicates which variable contains what value. The transition relation Δ transfers an EFSM from one state with variables containing certain values to another state in which the variables contain some other values.

Also note that the variable τ does not exist in a specification. It is merely a convenient way to unify input, output, and assignment actions. When an input event (say $G?x$) occurs, it is viewed as an assignment statement $\tau := (G?x)$ being executed. Similarly, an output action $G!e$ can be treated as an assignment action $\tau := (G!e)$. The value "$G?x$" or "$G!e$" can be a data structure that contains three fields indicating the gate, the variable to input or the expression to output, and a bit indicating whether input or output is performed.

A formal definition of the EFSM in Figure 6 is shown in Figure 7. The value space K includes $\{0, 1\}$ for variable A and B, $\mathcal{M}$ for variable M, and $\mathcal{I}$ and $\mathcal{O}$ for the special variable τ. The status set R contains all the possible combination of valid values for the variables in V. An element in Δ is denoted as "$(q_i, \rho_i) \rightarrow (q_j, \rho_j)$," which means a transition transfers the EFSM from state q_i with status ρ_i (where $\rho_i \in R$) to state q_j with status ρ_j. Recall that a status is a function maps a set of variables to a set of values; therefore, as an example, $\rho(B)$ represents the value of B in status ρ.

2.4 Other Extended Models

There are other exteded models used for protocol specification, such as Petri-Net, ETG, SDL, Estelle, LOTOS, Promela, and other programming language models. They are only different syntactically. In fact, it is not difficult to prove that all these extended models are *Turing equivalent*, which means that they can specify all

1. $Q = \{a, b, c, d, e, f\}$
2. $s = a$
3. $E = \emptyset$
4. $V = \{A, B, M, \tau\}$
5. $K = \{0, 1\} \cup \mathcal{M} \cup \mathcal{I} \cup \mathcal{O}$, where
   - $\mathcal{M}$ is a set of possible messages,
   - $\mathcal{I} = \{(U?M) \mid M \in \mathcal{M}\} \cup \{(R?A) \mid A \in \{0, 1\}\}$, and
   - $\mathcal{O} = \{(R!(M, B)) \mid M \in \mathcal{M}, B \in \{0, 1\}\}$
6. τ, whose value can be one of the elements in $\mathcal{I} \cup \mathcal{O}$.
7. $R = \{A = 0, A = 1\} \times \{B = 0, B = 1\} \times$
   $\{\tau = t \mid t \in \mathcal{I} \cup \mathcal{O}\} \times \{M = msg \mid msg \in \mathcal{M}\}$
8. $\Delta = \{(a, \rho_1) \rightarrow (b, \rho_2), (b, \rho_1) \rightarrow (c, \rho_3), ...\}$, where
   $\rho_1 \in R_1$, where $R_1 = R$
   $\rho_2 \in R_2$, where $R_2 = \{\rho \mid \rho(B) = 0\}$
   $\rho_3 \in R_3$, where $R_3 = \{\rho \mid \rho(M) = msg \wedge \rho(\tau) = (U?M)\}$

Figure 7: A formal definition of the EFSM in Figure 6

the computable problems known today. The EFSM model discussed in the previous section has the simplest syntax among all the extended models. Therefore, it is used as the fundamental model in this disseration.

2.5 Axiomatic Semantics

The *axiomatic semantics* technique is usually used as a program verification tool in software engineering. It provides a set of *axioms*, each of which is a rule describing how a program status changes when a statement is executed. A program status is described by an *assertion*. An assertion is normally denoted as a boolean expression, called a *predicate*, quoted by a pair of braces ({ }). The predicate states the condition under which the status of the program must satisfy. An assertion before a statement is called the *precondition* of the statement, and the assertion after, is called a *postcondition*. A program is verified from the most generalized precondition, and applied the axioms statement by statement to obtain a more restricted postcondition. If the final postcondition is consistent with what is expected, the program is proven correct. Further information on this technique can be found in [45].

Formally, recall the definition of status in Definition 2.3.1 on page 19 that a status is a function that maps a set of variables to a set of values. An assertion can be defined as a set of statuses that satisfy a certain property indicated in the predicate[1]. An axiom states the change of assertions, and therefore is a function that transforms one assertion to another. The definitions of statuses, assertions, and axioms are listed as follows.

[1]This definition will be revised in Section 7.2.

Definition 2.5.1 *Let V be a set of variables and K be a finite or infinite set of values. Then,*

1. $\rho : V \mapsto K$ *is a status,*
2. $R = \{\rho \mid \rho$ *is a status*$\}$ *is an assertion, and*
3. $\mathcal{X} : R \mapsto R$ *is an axiom.*

Then, the relationship between assertions and predicates is defined as follows:

Definition 2.5.2 *An assertion R is denoted by $\{p\}$ if $R = \{\rho \mid \rho \Rightarrow p\}$, where p is a predicate.*

From Definition 2.5.2, The following corollary can be derived.

Corollary 2.5.3 *If R_1 and R_2 are two assertions such that $R_1 = \{p_1\}$ and $R_2 = \{p_2\}$, then*

1. $R_1 \cap R_2 = \{p_1 \wedge p_2\}$,
2. $R_1 \cup R_2 = \{p_1 \vee p_2\}$,
3. $\overline{R_1} = \{\neg p_1\}$, *and*
4. $R_1 - R_2 = \{p_1 \wedge \neg p_2\}$.

Proof. Let ρ be a status. Then, points 1 through 4 can be proven as following.

1. $$\begin{aligned} \rho \in R_1 \cap R_2 &\Leftrightarrow \rho \in R_1 \wedge \rho \in R_2 \\ &\Leftrightarrow \rho \Rightarrow p_1 \wedge \rho \Rightarrow p_2 \\ &\Leftrightarrow \rho \Rightarrow p_1 \wedge p_2 \\ &\Leftrightarrow \rho \in \{p_1 \wedge p_2\} \end{aligned}$$

$$\begin{aligned} 2.\quad \rho \in R_1 \cup R_2 &\Leftrightarrow \rho \in R_1 \vee \rho \in R_2 \\ &\Leftrightarrow \rho \Rightarrow p_1 \vee \rho \Rightarrow p_2 \\ &\Leftrightarrow \rho \Rightarrow p_1 \vee p_2 \\ &\Leftrightarrow \rho \in \{p_1 \vee p_2\} \end{aligned}$$

$$\begin{aligned} 3.\quad \rho \in \overline{R_1} &\Leftrightarrow \rho \notin R_1 \\ &\Leftrightarrow \rho \Rightarrow \neg p_1 \\ &\Leftrightarrow \rho \in \{\neg p_1\} \end{aligned}$$

$$\begin{aligned} 4.\quad R_1 - R_2 &= R_1 \cap \overline{R_2} \\ &= R_1 \cap \{\neg p_2\} \\ &= \{p_1 \wedge \neg p_2\} \end{aligned}$$

□

Before showing an example of axiomatic semantics, some notations need to be defined. As a transition of an EFSM is executed, the values of the variables change accordingly. The values are preserved by *versions*, which store different values of the same variable. For example, after statement $x := x+1$ is executed, variable x changes into a different value. To distinguish the new value from the old ones, operator ν (pronounced "new") is used to indicate which version of a variable is referred. Hence, νx denotes a newer version of x, and $\nu\nu x$ (or $\nu^2 x$) represents an even newer version of νx. In general, if x is denoted as $\nu^0 x$, $\nu^{i+1} x$ is a newer version of $\nu^i x$ for any integer i. Therefore, $\nu^{-1} x$ denotes an older version of x, and $\nu^{-2} x$ is an older version of $\nu^{-1} x$.

If p is an expression, p^x_y denotes replacing each occurrence of x in p with y. For example, if expression p is $\nu x = x + 1$, $p^x_{\nu x}$ denotes $\nu\nu x = \nu x + 1$ (which is $\nu^2 x = \nu x + 1$). Therefore, in general, $p^x_{\nu^j x}$ replaces each occurrence of $\nu^i x$ in p with $\nu^{i+j} x$ for any integer i and j. The notation $p^x_{\nu^i x}$ can be abbreviate as $p_{\nu^i x}$, and the notation $p_{\nu^{i_1} x_1, \nu^{i_2} x_2, \ldots, \nu^{i_n} x_n}$ is equivalent to $(\ldots((p_{\nu^{i_1} x_1})_{\nu^{i_2} x_2})\ldots)_{\nu^{i_n} x_n}$.

Figure 8 shows an example of how axiomatic semantic can be used to prove the correctness of a program. The proving process can either be forward or backward. Figures 8a and 8b, respectively, show the forward and the backward axioms of an assignment statement $x := e$ (where x is a variable and e is an expression). The forward axiom indicates that if p is the predicate of the assertion before statement $x := e$, the statement's postcondition will be $\{p_{\nu^{-1}x} \wedge x = e_{\nu^{-1}x}\}$. That is, whatever condition showed in the precondition will still hold in the postcondition, except that the x's in p will be outdated by one version (denoted $p_{\nu^{-1}x}$). In addition, the postcondition imposes a new condition, $x = e_{\nu^{-1}x}$, which means the new x is equal to the result of expression e, where every reference to the variable x gets an old value.

On the contrary, the backward axiom in Figure 8b assumes a postcondition first, and derives preconditions later. The axiom shows that if the postcondition satisfies predicate p, the precondition of the statement $x := e$ will preserves p, only that those x's referenced by p contains values from the future. Therefore, the x's must be marked as newer versions in order to distinguish from the value of x before the statement is executed (denoted $p_{\nu x}$). In addition, knowing that $x := e$ will be executed, the axiom also imposes a new condition, $\nu x = e$, to the precondition, which means the new x will be equal to the result of e if $x := e$ is executed.

Figure 8c is a simple program to illustrate how axiomatic semantics can prove that this program set the value of variable x to 1 under any circumstance. Figure 8d shows how the program is verified using the forward axiom in Figure 8a. The initial assertion of the program is $\{\mathbf{true}\}$ (line 1), meaning that no condition is required.

$\{p\}\ x := e;\ \{p_{\nu^{-1}x} \wedge x = e_{\nu^{-1}x}\}$

(a)

$\{p_{\nu x} \wedge \nu x = e\}\ x := e;\ \{p\}$

(b)

$x := 0;$
$x := x + 1;$

(c)

1. $\{\mathbf{true}\}$
2. $x := 0;$
3. $\{\mathbf{true} \wedge x = 0\}$
4. $\{x = 0\}$
5. $x := x + 1;$
6. $\{\nu^{-1}x = 0 \wedge x = \nu^{-1}x + 1\}$
7. $\{x = 1\}$

(d)

1. $\{\mathbf{true}\}$
2. $\{\nu^2 x = 1 \wedge \nu^2 x = \nu x + 1 \wedge \nu x = 0\}$
3. $x := 0;$
4. $\{\nu x = 1 \wedge \nu x = x + 1\}$
5. $x := x + 1;$
6. $\{x = 1\}$

(e)

Figure 8: An example illustrating the axiomatic semantics technique (a) A forward axiom of an assignment statement, (b) A backward axiom of the same statement, (c) A simple program, (d) Proving the program using forward axioms (forward evaluation), and (e) Proving the same program using backward axioms.

Applying the forward axiom to the assignment statement in line 2, a new assertion in line 3 is obtained. The assertion implies the assertion in line 4, and applying the same axiom to the statement in line 5 generates the assertion in line 6. Finally, line 6 implies line 7 and the final value of x being one is proven.

The same argument can be proven using the backward axiom in Figure 8b. In Figure 8e, the proving process starting with the assumption that the resulting value of x is one (line 6). Applying the backward axiom to the statement in line 5 will result in the assertion in line 4. Applying it again to line 3 derives the assertion in line 2, which can be implied by the assertion in line 1. Since the assertion in line 1 imposes no restriction, the final value of x will indeed be one regardless what initial condition is. The proving process used in Figure 8d is called a *forward evaluation*, and the process used in Figure 8e is called a *backward evaluation*.

Axiomatic semantics is the fundamental concept of this dissertation. All the proposed method described in Chapters V, VI, VII, and VIII are all based on this technique.

CHAPTER III

Test Case Generation for Finite State Machines

In the past few years, researchers have proposed numerous TCG methods based on the FSM model. Some of these methods are presented and arranged into three sections in this chapter. The first section (Section 3.1) introduces TCG methods without *test purposes*. A test purpose states what type of fault a test case tries to detect. In Section 3.2, some TCG methods that considers test purposes are introduced. These methods give each test case a responsibility to detect the error of a single transition. Section 3.3 introduces some methods designed for optimizing the length of a test suite. These methods try to combine several test cases to eliminate overlapped or redundant subsequences.

Since the FSM in Figure 4 is not an interesting example for most of the TCG method described in this chapter, Figure 9, which is borrowed from Sidhu's paper [52], is used instead. The input actions in Figure 9 are abbreviated as A and B, and the output actions are represented by 0 and 1. Note that state a does not accept input B. It is assumed that when the FSM is at state a and receives an input B, the FSM remains in the same state and outputs nothing. It looks like having a transition loops from state a back to a with a label B/-, where "-" represents an empty output.

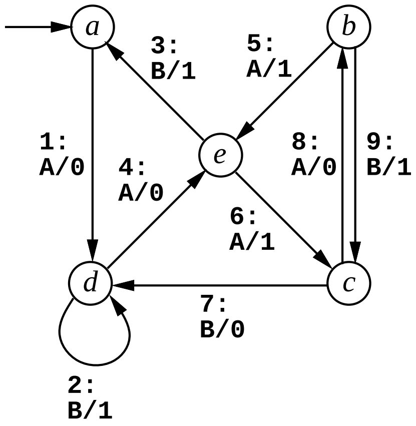


Figure 9: A sample FSM

3.1 Methods without Test Purposes

Both R- and T-methods discussed in this section do not divide a test suite into test cases. They generates only one test event sequence; and therefore, a test suit and a test case can be used interchangeably in this section.

3.1.1 The R-Method

The R-method stands for random walk. It is the simplest among all introduced in this chapter. The state transition graph of an FSM is traversed by randomly selecting one next possible transition, until every transition is traversed at least once. For example, a path selected by the R-method for the FSM in Figure 9 can be:

$$\langle 1, 2, 2, 4, 3, 1, 4, 6, 7, 2, 4, 3, 1, 4, 6, 8, 5, 6, 8, 9 \rangle$$

The test case is the I/O sequence generated by the path, which inputs

$$\langle rABBABAAABBABAAAAAAAB \rangle$$

and expects

$$\langle \text{-0110100101010100101101} \rangle$$

where the r represents the reset signal.

Although studies has found that, *statistically*, test cases generated by the R-method provide acceptable fault coverage [37, 61], it has the following problems.

1. Nothing is guaranteed in a test case. Whether a test case detects any error depends on how lucky you are.

2. A test case may contain many redundant parts. That is, some paths that have already been traversed may still be repeated for several times, which makes a test case unreasonably long.

3. No test case serves any purposes. That is, no one is able to know what type of fault a test case can detect.

Therefore, it is necessary to have more control over the generation of test cases.

3.1.2 The T-Method

To eliminate problem 2 of the R-method, a little control in path traversing is introduced. The T-method [41] finds the shortest *transition tour* from the state transition diagram as a test case. The method tries to find a *Euler tour* ,which traversed every transition exactly once. If a Euler tour does not exist, a minimal amount of transitions are duplicated to create one.

For example, the FSM in Figure 9 does not have a Euler tour. Hence, transitions 4, 6, and 8 are duplicated (Figure 10) to create a Euler tour. The dashed arrows are added such that

$$\langle 1, 2, 4, 6, 8, 9, 7, 4, 6, 8, 5, 3 \rangle$$

forms a Euler tour. The test case is to input $\langle rABAAABBAAAAB \rangle$ and to expect ⟨-010101001011⟩.

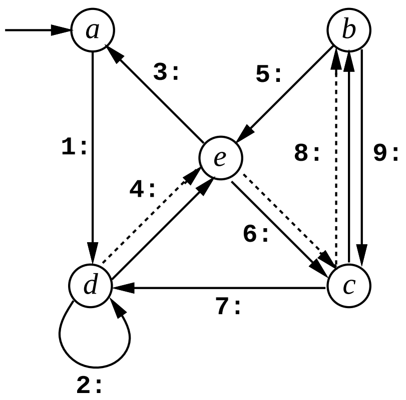


Figure 10: Augmented graph on Figure 9

A test case generated from the T-method still suffers from not having any test purposes. What error can be detected by traversing every transition is unclear. Similar to the R-method, T-method cannot guarantee the absence of any type of fault. Moreover, since the R-method may traverse a more complicated path than the T-method does, it is very likely that R-method has a better fault coverage than the T-method, which makes T-method less attractive.

3.2 Methods with Test Purposes

TCG methods providing no test purposes have a difficulty to convince anyone that the test cases generated detects any error. As a result, methods considering test purposes are proposed. In the TCG methods introduced in this section, each test case generated by test methods is in charge of the correctness of a single transition. When a protocol implementation passes a test case, one can have the confidence that the transition corresponding to the test case is likely to be correct.

The methods presented in this section generally follows these steps.

1. Test each transition individually.

2. For each transition tested, observe whether

   (a) The output of the transition is correct, and

   (b) The tail state of the transition is correct.

Hence, a test case contains three parts.

1. The *preamble*: A path from the initial state to the head state of the transition being tested.

2. The transition: At this part, check if the output of the transition is correct.

3. The *postamble*: A path used to check the correctness of the transition's tail state.

For a transition in an FSM, to determine the correctness of the output is trivial, but to decide whether the transition enters a correct tail state is not. Therefore, methods such as the D-, U-, and W-methods mainly concern about how to confirm the correctness of the tail states, and propose different ways to construct the postamble.

Among the methods proposed, the U-method (or the UIO method) is the most referenced approach. It has also been shown to yield good fault coverage [52]. Several varieties of the U-method are proposed to improve its fault coverage and test case length. These methods are also introduced in this chapter.

3.2.1 The D-Method

The D-Method [17] uses so called *distinguishing sequence* to ensure the correctness of a transition's tail state. An input sequence is called a distinguishing sequence for an FSM if feeding it into the FSM will make the FSM produce different output sequence for different state. For example, input sequence $\langle BB \rangle$ is a distinguishing sequence for the FSM in Figure 9 because the FSM produces different output at different state when given input sequence $\langle BB \rangle$. Table 1 shows the output of the input string $\langle BB \rangle$ at different state. Note that none of the state produces the same output.

Therefore, the test case for transition 1 is through path $\langle 1, 2, 2 \rangle$, with input events $\langle rABB \rangle$ and output events $\langle$-011$\rangle$. The first input, r, is the preamble of the test case, the second is the tested transition itself, and the final two inputs are the distinguishing sequence $\langle BB \rangle$, which ensures the FSM goes to state d if the FSM outputs $\langle 11 \rangle$ (since no other state in the FSM has such output in responding to input sequence $\langle BB \rangle$).

Table 1: Output of sequence $\langle BB \rangle$ at every state.

State	Path	Output
a	- -	- -
b	9,7	10
c	7,2	01
d	2,2	11
e	3,-	1-

Combining the test cases, the final test suite for the FSM will be

$$\langle rABB, rABBB, rAABBB, rAABB, rAAAAABB, ...\rangle$$

Showing here is the test cases for transitions 1 to 5.

The drawback of the D-method is that not every FSM has a distinguishing sequence, in which case the method fails to generate any test cases. Therefore, some criteria are relaxed in the following methods to construct some sequences that are similar to the distinguishing sequence, yet still able to check the correctness of the tail state.

3.2.2 The U-Method

Instead of using a single sequence to distinguish all state, the U-method (or UIO-method) uses a sequence for each state in an FSM [49]. This sequence is called a *Unique I/O (UIO)* sequence, which cannot be duplicated by any other state in the graph. Such a sequence can be used as the postamble of a test case to ensure the correctness of a tail state. Observing the UIO for a state reveals that the FSM originates from that state.

The states in Figure 9 and their UIOs are shown in Table 2. Each state in the table has an unique I/O event sequence that is not exhibited by any other state. The test case for a transition is composed of a preamble, the transition itself, and the UIO of the tail state of the transition. For example, the test case for transition 1 is $\langle r/\text{-}, A/0, B/1, B/1\rangle$, which traverses through path $\langle 1, 2, 2\rangle$

Table 2: The UIOs for the states in Figure 9

State	Path	UIO
a	-	B/-
b	5,6	A/1, A/1
c	7	B/0
d	2,2	B/1, B/1
e	6,8	A/1, A/0

The test cases generated by the U-method for the FSM in Figure 9 is shown in Table 3. A test case shown in the table is divided into two columns; the preamble lead to the head state of the transition being tested, and the transition and the UIO of its tail state confirms the correctness of the transition. The path traversed by a test case is shown in the fourth column, in which a blank is used to separate the path of preamble and the path of the rest transitions in the test case. A test suite can be constructed by concatenating these test cases, such as

$$\langle rABB, rABBB, rAABB, rAAAA, rAAAAAAA, ...\rangle$$

Table 3 will be used as an example to illustrate the test suite optimization methods introduced in Section 3.3.

Table 3: The test cases generated by the U-method

Transition	Preamble	Transition & UIO	Path of Test Case	Tail State
1	r	ABB	r 122	d
2	rA	BBB	r1 222	d
3	rAA	BB	r14 3-	a
4	rA	AAA	r1 468	b
5	$rAAAA$	AAA	r1468 568	b
6	rAA	AB	r14 67	d
7	$rAAA$	BBB	r146 722	d
8	$rAAA$	AAA	r146 856	c
9	$rAAAA$	BB	r1468 97	d

There are two major drawbacks for the U-method.

1. Some state may not have a UIO. The method fail to find any test case for the incoming transitions of a state if the state does not have an UIO.

2. An I/O sequence, which is unique for a state in the specification, does not necessarily mean that it is unique for the same state in the implementation.

Therefore, some improvements of the U-method are proposed and are introduced in Section 3.2.3.

3.2.3 The UIOv- and UIOpv-Methods

As mentioned above, a UIO may not retain its uniqueness in an erroneous implementation. It is likely that an error in the implementation causes another state to generate an I/O sequence similar to other states' UIO. The example in Figure 11 is redrawn from Chan's example in [8]. In the original specification (Figure 11a), state c has a UIO $\langle B/1, A/1 \rangle$. However, in the faulty implementation (Figure 11b), the erroneous transition (the white arrow) makes state a accepts I/O sequence $\langle B/1, A/1 \rangle$. Therefore, states a and c are indistinguishable by the UIO of state c. Moreover, since detecting the faulty transition relies on the UIO of state c, the error becomes undetectable by the U-method.

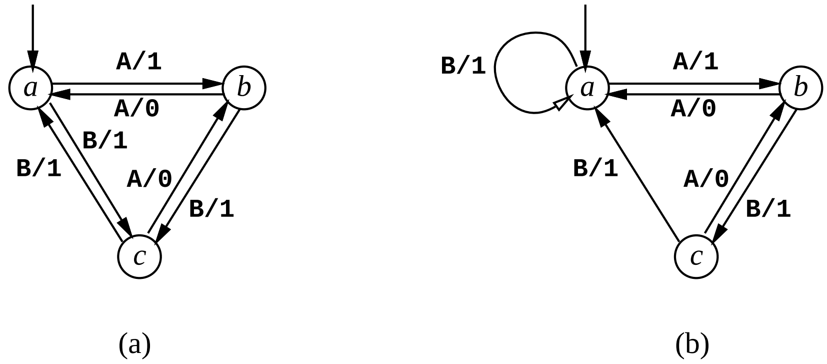


Figure 11: An UIO that becomes nonunique in an implementation, (a) the specification, (b) an implementation in which the UIO of state c can be reproduced by state a.

To resolve the problem, the UIOv-method [8] proposes a verification phase before applying the test cases generated by the U-method. The idea is to check every state in the implementation to ensure the UIOs are really unique in the implementation. For example, to guarantee the uniqueness if the UIO of state c in Figure 11, sequence $\langle B/1, A/1 \rangle$ is applied to states a and b; i.e., sequence $\langle rBA, rABA \rangle$ is applied. If the

UIO remains unique in the implementation, the subsequence $\langle BA \rangle$ should not output a $\langle 11 \rangle$. (It does output a $\langle 11 \rangle$ in Figure 11b, so the error is detected.)

However, the "nonunique UIO" problem shown in Figure 11 seldom happens. Solving this problem by verifying the UIO of every state against every other states is not cost-effective. (This requires $n(n-1)$ extra test cases if n is the total number of states.) Hence, Yu et al. has found the condition in which nonunique UIO can possibly happen [64]. Only under this condition that a verification is necessary. The method is called the UIOpv-method, where "pv" stands for partial verification.

The condition is that when there are other transitions in an FSM that have the same I/O label as the first transition of an UIO, the UIO needs to be verified. Moreover, it is unnecessary to verify the UIO against every other states; only those states that have an outgoing transition with the same I/O label as the UIO's first transition need to be checked. For example, in Figure 11a, UIO $\langle A/1 \rangle$ for state a does not need to be verified. However, the UIO $\langle A/0, A/1 \rangle$ for state b needs to be verified against state c because state c has an outgoing transition labeled $A/0$.

In addition of solving the nonunique UIO problem, both UIOv- and UIOpv-methods can also generate test cases for states without any UIOs. The method used is similar to the Wp-method and will be introduced in Section 3.2.5.

3.2.4 The W-Method

When there is no distinguishing sequence in an FSM, the D-method fails. To relax this requirement, the W-method [11] uses a set of sequence (instead of just one) to distinguish the states. The set of sequence is called a *characterizing set*. Each sequence in the characterizing set, individually, may not distinguish every state in an FSM, but their combined outcome, however, is able to tell one state from another.

For example, Table 4 shows the output sequences of the FSM in Figure 9 for a characterizing set $\{\langle AA\rangle, \langle B\rangle\}$. Individually, $\langle AA\rangle$ cannot distinguish state c from state d, and neither can $\langle B\rangle$ distinguish states b, d, and e. However, by combining both sequences, all five states can be distinguished. To test transition 1, both $\langle rAAA\rangle$ and $\langle rAB\rangle$ must be applied. The former used sequence $\langle AA\rangle$ as a postamble and the latter uses $\langle B\rangle$. Therefore, a test suite may look like the following (showing here only the test cases for transition 1 to 5).

$\langle rAAA, rAB, rABAA, rABB, rAABAA, rAABB, rAAAA, rAAB, rAAAAAAA, rAAAAAB, ...\rangle$

Table 4: The outputs of input sequences $\langle AA\rangle$ and $\langle B\rangle$ at each state

State	AA	B
a	00	-
b	11	1
c	01	0
d	01	1
e	10	1

3.2.5 The Wp-Method

The "p" of the Wp-method [15] stands for "partial," meaning that instead of using a whole characterizing sets for all states, the Wp-method only uses part of the characterizing set for a state. For example, from Table 4, one can observe that states a, b, and e can be distinguished by $\langle AA\rangle$ alone. Applying the second sequence $\langle B\rangle$ is unnecessary for these states; and therefore, the sequence can be eliminated. On the other hand, states c and d cannot distinguish each other by just $\langle AA \rangle$, so they need to be distinguished by the second sequence, $\langle B \rangle$. Such a partial characterizing set for each state is called the *signature* of the state. The signatures of the states in Figure 9 is shown in Table 5.

Table 5: Signatures of the states in Figure 5

State	Signature
a	$\{\langle A/0, A/0 \rangle\}$
b	$\{\langle A/1, A/1 \rangle\}$
c	$\{\langle A/0, A/1 \rangle, \langle B/0 \rangle\}$
d	$\{\langle A/0, A/1 \rangle, \langle B/1 \rangle\}$
e	$\{\langle A/1, A/0 \rangle\}$

Both the U- and W-methods can be viewed as a relaxed version of the D-method. While the D-method uses *one* sequence to distinguish *all* states, the U-method uses *one* sequence to distinguish *one* state from any other states, and the W-method uses *multiple* sequences to distinguish *all* states. The Wp-method is a general solution for all three. When every signature contains only one sequence, the Wp-method reduced to the U-method. When all the signatures contain the same input sequences, the Wp-method is equivalent to the W-method. If both of the above are true (i.e., all signatures contain only one sequence, and all their inputs are the same), the Wp-method becomes a D-method.

3.3 Test Suite Optimization

As one can observe, the test suite shown in the previous sections are full of redundancies. For example, in Table 3, after transition 1 is tested, the FSM is located at state *d*. In the original U-method, one has to reset the FSM and traverse through the preamble to test transition 2. Alternatively, one can omit the preamble and start testing transition 2 once transition 1 is tested.

Moreover, test case for transitions 1 is contained in the test case of transition 2, which means if transition 2 is tested, so is transition 1. Therefore, having a test case for transition 1 is unnecessary. If the overlapped area of the test cases can be combined, the overall length of the test suite can be much more shorter.

Hence, test suite optimization methods are proposed to minimize the length of the test suite. Although the test suite optimization methods introduced in this section are based on the U-method, they do not have to. In fact, these methods can be used as tools to combine test cases into a shorter test suite, no mater what TCG method creates those test cases. The test case in Table 3 will be used as an example in the rest of this section. However, the test cases mentioned here contains only the transitions and their UIOs. The preamble are to be removed in the optimized TCG methods, and are not considered part of the test cases.

3.3.1 The Chinese Postman Problem

The methods introduced in this chapter are all related to a problem, called the *Chinese Postman Problem* [34]. The problem is to find a *postman tour* in a directed and weighted graph, such that each edge in the graph is traveled at least once and the total of the weights (called cost) in the tour is minimum. For example, in Figure 12a, tour $\langle abecdecdcbea \rangle$, which has total cost of 38, is a postman tour.

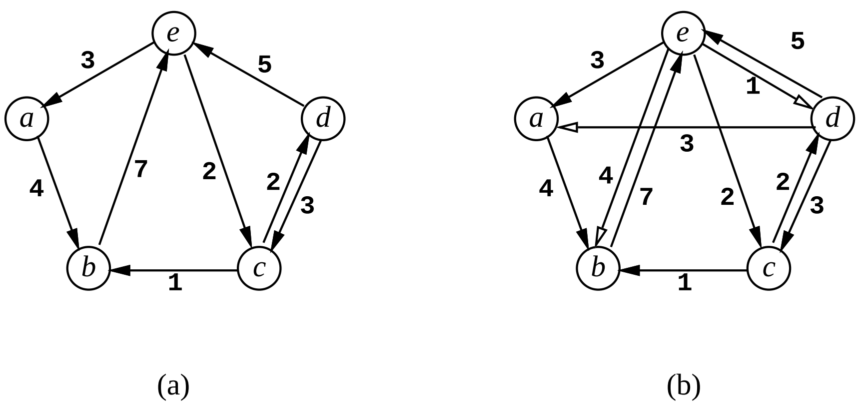


Figure 12: (a) The Chinese Postman Problem and (b) The Rural Chinese Postman Problem

The *Rural Chinese Postman Problem* is a variety of the Chinese Postman Problem, in which some edges are considered "short cuts" and can be traversed when necessary. For example, in Figure 12b, the white arrows are the optional edges. A postman tour of this graph traverses $\langle abecdedcbea \rangle$ and has a total cost of 33. Note that not every white arrows needs to be traveled.

While efficient polynomial algorithms have been found for Chinese Postman Problem [14, 16], it has been proven that Rural Chinese Postman Problem is NP-hard in the most general case [38]. Since the methods introduced in this section all reduce the test suite optimizing problem to the Rural Chinese Postman Problem, none of them have efficient way to generate minimum length test suite. It has also been shown that the fault coverage of the test suite becomes worse after the test suite is optimized [65].

3.3.2 Aho's Method

Aho et al. first transform the test suite optimization problem to the Chinese Postman Problem [1, 2]. The method is based on the idea that after a test case is finished, it is not always necessary to reset the FSM to test another transition.

The proposed method creates a directed and weighted graph similar to the original FSM, but with the following changes:

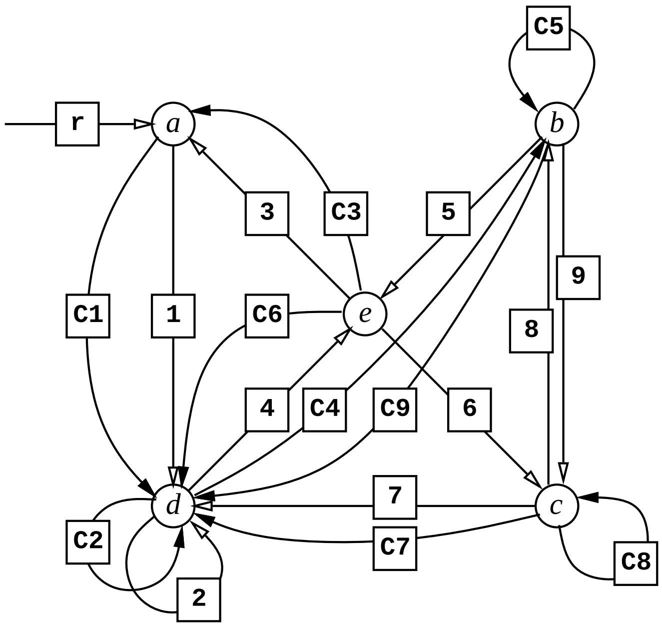
1. Make the original transitions optional edges, and associate each transition with cost of one.

2. For each transition with the UIO of its tail state, add a mandatory edge from the head state of the transition to the state where the UIO ends. Associate the added edge with a cost that is equivalent to the length of the UIO plus one.

The FSM in Figure 9 with test cases in Table 3 is modified as the directed graph shown in Figure 13. The white arrows in the graph represents the optional edges. For example, the test case for transition 1 is represented by edge C1, which ends at state *d*, where the original test case ends. A postman tour found from the graph represents a optimized test suite.

For example, a postman tour of Figure 13 can be

$$\langle C1, C2, 4, C6, C4, C5, 9, C8, C7, 4, 6, 8, C9, 4, C3 \rangle$$



1: A/0	4: A/0	7: B/0
2: B/1	5: A/1	8: A/0
3: B/1	6: A/1	9: B/1

C1: A/0,B/1,B/1	C6: A/1,B/0
C2: B/1,B/1,B/1	C7: B/0,B/1,B/1
C3: B/1,B/-	C8: A/0,A/1,A/1
C4: A/0,A/1,A/0	C9: B/1,B/0
C5: A/1,A/1,A/0	

Figure 13: The graph modified from Figure 9 using Aho's method

and the optimized test suite is

$$\langle (A/0, B/1, B/1), (B/1, B/1, B/1), (A/0), (A/1, B/0), \ldots \rangle$$

Since every transition plus the UIO of its tail state is represented by a mandatory edge, each transition is guaranteed to be tested. Between every two test cases, any path can be traversed as long as the cost can be minimized. The resulting cost of a postman tour is the length of the final test suite.

3.3.3 Shen's Method

Sometimes, a state can have more than one UIO. Using some of the UIOs may obtain a shorter test suite than the others. Therefore, while Aho et al. consider only one UIO per transition, Shen et al. try to utilize multiple UIOs [51]. For example, Table 6 shows multiple UIOs of the FSM in Figure 9. Any UIOs of a state can be selected in a test case.

Similar to Aho's method, this method augments the original FSM to create a graph for postman tour. The changes are as follows.

1. Change the original transitions into optional edges.

2. For each transition, add an auxiliary state.

3. Add a mandatory edge from the head state of the transition to the auxiliary state and set the cost of the edge to one.

4. For each UIO of the tail state of the transition, add an optional edge from the auxiliary state to the state where the UIO ends. Set the cost of the edge as the

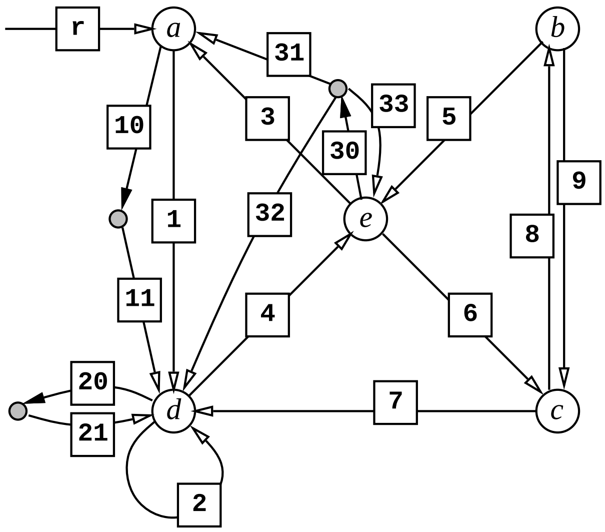
Table 6: Multiple UIOs for the FSM in Figure 6

State	UIO(s)	Tail State
a	B/-	a
	A/0,B/1	d
	A/0,A/0	e
b	A/1,A/1	c
	A/1,B/1	a
	B/1,B/0	d
c	B/0	d
	A/0,A/1,B/1	a
d	B/1,B/1	d
e	A/1,A/0	b
	A/1,B/0	d
	B/1,B/-	a
	B/1,A/0	d

length of the UIO.

The resulting graph modified from Figure 9 is shown in Figure 14. In the figure, the shaded circles are the auxiliary states. Only the auxiliary states for transitions 1, 2, and 3 are shown; the rest of the transitions can be derived by the same manner. State a, for example, has three UIOs; and therefore, transition 3 has an auxiliary state with three outgoing transitions pointing to states a, d, and e, which are the tail state of the UIOs.

Since the incoming transition of every auxiliary states are mandatory, the auxiliary state must be selected. Hence, at least one of the UIO must be traversed. As



1: A/0	4: A/0	7: B/0
2: B/1	5: A/1	8: A/0
3: B/1	6: A/1	9: B/1

10: A/0	20: B/1	30: B/1
11: B/1,B/1	21: B/1,B/1	31: B/-
		32: A/0,B/1
		33: A/0,A/0

Figure 14: The graph modified from Figure 9 using Shen's method

the result, each transition will be tested. A postman tour of the augmented graph represents a optimized test suite.

3.3.4 Chen's Method

As mentioned in the beginning of this chapter, test cases are often overlapped. For example, after traversing the test case of transition 8 (path $\langle 856 \rangle$), the first two transitions of the test case for transition 5 ($\langle 568 \rangle$) has been traversed. Therefore, traversing path $\langle 8568 \rangle$ tests both transitions 5 and 8.

Chen et al. are among those who first take advantage of the overlapping test cases [10]. The method creates a directed graph based on the following steps.

1. Change all the transitions in the original diagram into optional edges.

2. For each transition T, add two auxiliary states h_T and t_T.

3. Add a mandatory edge from h_T to t_T. The cost of the edge is the length of the test case for transition T.

4. Add two optional edges: one from the head state of T to h_T and the other from the t_T to the state where the UIO of transition T's tail state ends. Each costs zero.

5. For every other transition T' in the graph, add an edge from t_T to $h_{T'}$. The cost associated with the edge is decided as follows:

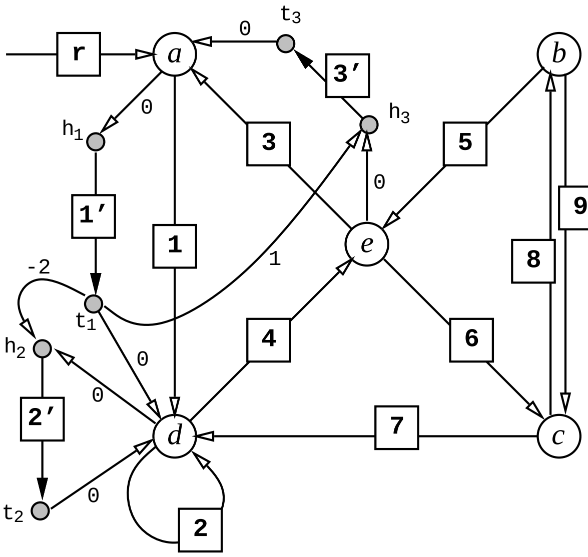
   (a) If the test cases of T and T' cannot be overlapped, the cost is the length of the shortest path from the state where the test case of transition T ends, to the head state of transition T'.

(b) If the test cases of T and T' can be overlapped, the cost is a negative number whose absolute value is the number of transitions being overlapped.

For example, in Figure 15, transitions 1, 2, and 3 have auxiliary states h_1 and t_1, h_2 and t_2, and h_3 and t_3, respectively. The transition $1'$ is a mandatory transition whose cost is three, wrt. the test case of transition 1. Since the test cases for transitions 1 and 2 overlap by two events, the edge pointing from t_1 to h_2 cost -2, meaning that when transition 2 is tested immediately after transition 1, two events are saved. On the other hand, if transition 3 is tested after transition 1, one extra transition (transition 4) must be traversed to reach the head state of transition 3. Therefore, the edge from t_1 to h_3 has a cost of one. For clarity, only some auxiliary states and transitions are shown in Figure 15; other states and transitions can be added with the same manner.

Since the edges from auxiliary states h's to t's are mandatory, all transitions will be tested. In addition, the negative costs of the edges encourage overlapping. The total cost of a postman tour in the augmented graph is the length of the test suite.

It is found that heuristic methods that solves *maximum cardinality minimum cost matching problem* [46] can be used to find a shortest path that traverse through all the mandatory edges generated by the method. Chen et al. observes that traversing a mandatory edge E_2 immediately after traversing mandatory edge E_1 is equivalent to matching the tail state of E_1 with the head state of E_2. The cost of matching the two states is the cost of the edge between them. The cost between the head and the



1: A/0	**4: A/0**	**7: B/0**
2: B/1	**5: A/1**	**8: A/0**
3: B/1	**6: A/1**	**9: B/1**

1′: A/0,B/1,B/1
2′: B/1,B/1
3′: B/1,B/-

Figure 15: The graph modified from Figure 9 using Chen's method

tail states of the same mandatory edge is set to be infinite. If a minimum match of the states forms a single tour, a postman tour is found and so is the optimal solution. Otherwise, several subtours are combined using the procedure that solves the Rural Chinese Postman Problem. Since mandatory edges are packed into subtours, the complexity of finding a postman tour will be reduced.

3.3.5 Miller's Method

Sometimes, it is interesting to find out that things usually come back to the basic. With a clever observation, Miller et al. found that under certain condition, the transition tour generated by T-method is good enough for an optimized test suite.

In [40], Miller defines *converging edges* as two or more transitions with the same label pointing to the same tail state. An FSM is *definitely diagnosable* if it does not possess any converging edges. In a definitely diagnosable FSM, it can be proved that every suffix of a path plus the UIO of the last state of the path, is a UIO of the head state of the suffix. This statement is proved by induction as follows.

Theorem 3.3.1 *Let p be a path of a definitely diagnosable FSM, and q be path p plus the UIO of the last state of p. If s_i is a state in p and q_i represent a suffix of q whose head state is s_i, then q_i is a UIO of s_i.*

Proof. Let there be n states in p.

Basis: Since q_n is the UIO appended after the transition tour, the statement holds.

Hypothesis: Suppose that q_{n-k} is a UIO of state s_{n-k}.

Induction: Since there is no converging edges in the FSM, there is no other state that has an outgoing transition that points to s_{n-k} and has the same label as the transition from state $s_{n-(k+1)}$ to s_{n-k}. Because q_{n-k} is a UIO of state s_{n-k}, $q_{n-(k+1)}$ is a UIO of state $s_{n-(k+1)}$.

□

Thus, a transition tour with the UIO of the last state is equivalent to overlapping all the test cases generated by the U-method.

If an FSM is not definitely diagnosable, the converging edges are removed. The resulting graph may be partitioned into several pieces, but since converging edges have been removed, each of the piece are definitely diagnosable. Then, a transition tour or path[1] is found for each partition. The tours, paths and the converging edges form a set of segments and can be linked together using the maximum cardinality minimum cost matching algorithm described in Section 3.3.4.

Miller's method is the best optimizing algorithm known so far. However, as pointed out by [65], optimizing the test suite often sacrifices the fault coverage. Although not being formally proved, it is accepted that usually, the shorter the test suite, the worse the fault coverage is.

3.4 Summary

In this chapter, TCG methods for FSMs are introduced. The R- and T-methods attempt to traverse all transitions in an FSM at least once. However, they fail to point

[1] Since a subgraph may not be strongly connected, it is possible that no transition tour exists. In such cases, a *transition path* whose tail state and head state are not the same is acceptable.

out what errors their test cases can detect. While the D-method uses a distinguishing sequence for all states in an FSM, the U-method uses a unique I/O sequence for each state to determine whether a state has been reached. The D- and U-methods are designed for *single-transition faults*, which assumes only one transition in an implementation can go wrong. The UIOv- and UIOpv-methods improves the U-method to cover a wider range of faults. The W-method, on the other hand, uses a characterizing set to distinguish the states. The W-method has been proved to be able to achieve full coverage if the number of states in the implementation is no larger than the number of states in the specification. The Wp-method improves from the W-method to obtain a shorter test suite. The UIOv-, UIOpv-, W- and Wp-methods are believe to have the same fault coverage. However, the length of the test suite can be too long to have any practical uses.

Several test suite optimization methods are also introduced. Aho at el. first incorporate Rural Chinese Postman Problem to the area of test suite optimization. The method proposed eliminates unnecessary preamble for the test cases. Shen and his colleagues utilize multiple UIO and uses whatever UIOs necessary for test cases as long as they make the test suite shorter. Chen et al. find out that test cases can be overlapped and proposes a method that makes the test suite even shorter. Miller and Paul discover the condition in which a transition tour is as good as overlapped UIO test sequences. As the result, combining the T- and U-methods and the heuristics procedure used in Chen's method, they propose the best test suite optimization method so far.

The FSM is a simple abstract model, but is too simple to specify many real-life protocol. Thus, even after the research of TCG methods for FSMs has been developed for years, test cases are still derived manually. In fact, testing a real-life protocol often need to deal with both control and data aspects. Since FSM cannot express the latter, more sophisticated specification models must be used. Therefore, in the next chapter, some pioneer work for Estelle and LOTOS are discussed.

CHAPTER IV

TCG for Extended Models

4.1 Why Extended Models?

With well established TCG methods for FSM, it is often asked why TCG methods for extended models are needed. A short answer is that *theoretically*, extended models are Turing equivalent; therefore, the problems that can be specified by extended models are not necessarily able to be defined in terms of FSM. However, some may argue that those extended models introduced in Chapter II are more powerful than FSM only if the values allowed for the variables are unbounded. In practice, the computers used today cannot have unbounded variables, and thus are all FSMs. So, why should extended models be used?

Consider the sender side of a Go-Back-N protocol specified by an EFSM shown in Figure 16. A Go-Back-N protocol has a window size W. A sender can send up to $W-1$ messages continuously to a receiver without receiving any acknowledgements. Each message is associated with a cyclic sequence number between 0 and $W-1$. The receiver acknowledges the receiving of the messages by sending an acknowledgment with the sequence number of the last message it has correctly received. The acknowledgment acknowledges all the messages that has been received before the last received message. When no acknowledgment has been received for a certain amount of time, a timeout signal is triggered and the sender resends those messages that have not yet been acknowledged. Note that in Figure 16, the window size W is received from the upper layer user of the protocol.

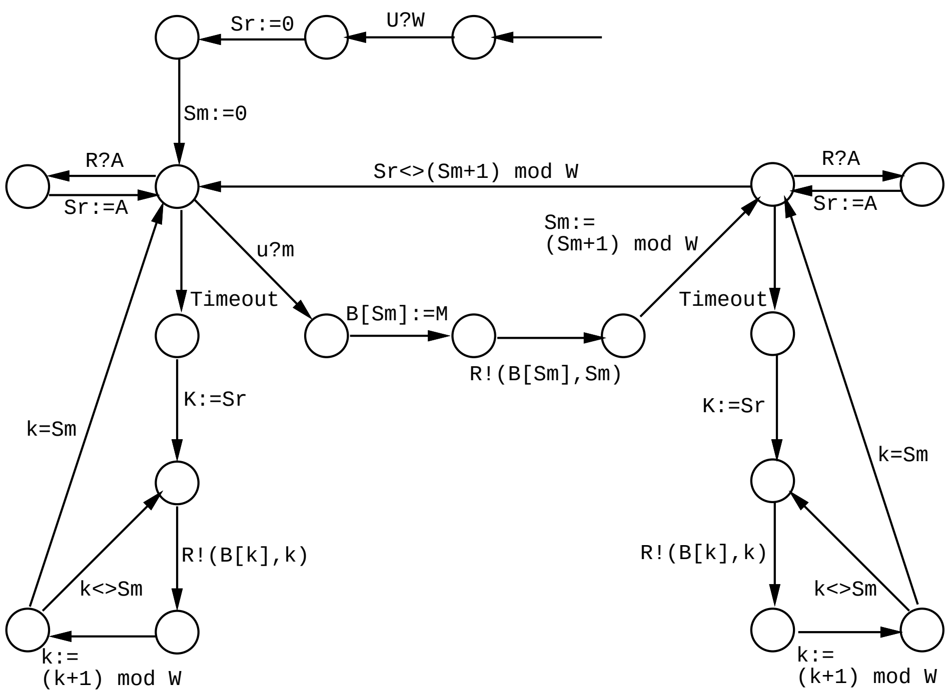


Figure 16: An EFSM specification for Go-Back-N Protocol

Now, consider specifying the same protocol using FSM model (Figure 17). It requires 28 states just to specify the case of $W = 4$. When W increases, the number of states increases in $O(W^3)$. If the upper bound of W is n, the total number of states needed to specify the protocol is in $O(n^4)$. Using the U-method, an average test case can contain $O(n)$ transitions. Therefore, the total length of a test suite boosts to

$O(n^5)$, and still cannot detect many types of errors due to the U-method's limitation. Just try to imagine the improved UIOv-method, which must verify every UIO against every other state. The resulting test suite can be as long as $O(n^9)$! The length of the test suites for W- and Wp-methods are estimated to be similar to the figure of the UIOv-method.

Using the EFSM model in Figure 16, the number of states will not increase with the value of W; no matter how large W is, the same diagram can still specify the protocol. Although a transition needs to be tested several times for different variable values, it is often unnecessary to exhaustively test every possible value.

There is an assumption called the "competent programmer assumption," meaning that a programmer will not write a program arbitrarily. The implementation and the specification will very much look alike, except for some small discrepancies caused by human errors. The problem of flattening an extended model into a FSM is that the semantics of the variables is often lost in the maze of the FSM's transitions. For example, a loop in an EFSM will be unfolded into a sequence of transitions in FSM. As most experienced programmers realize, a boundary condition and a loop invariant are quite adequate for testing the correctness of the loop. However, such phenomenon cannot be seen from the sequence of transitions in the FSM. Therefore, every transition seems equivalently important in an FSM and all need to be tested.

Unfortunately, testing an extended model remains a challenging problem. Little research has been done and existing solutions are far from satisfactory. The reason is that to test an extended model, one has to consider both control and data aspects

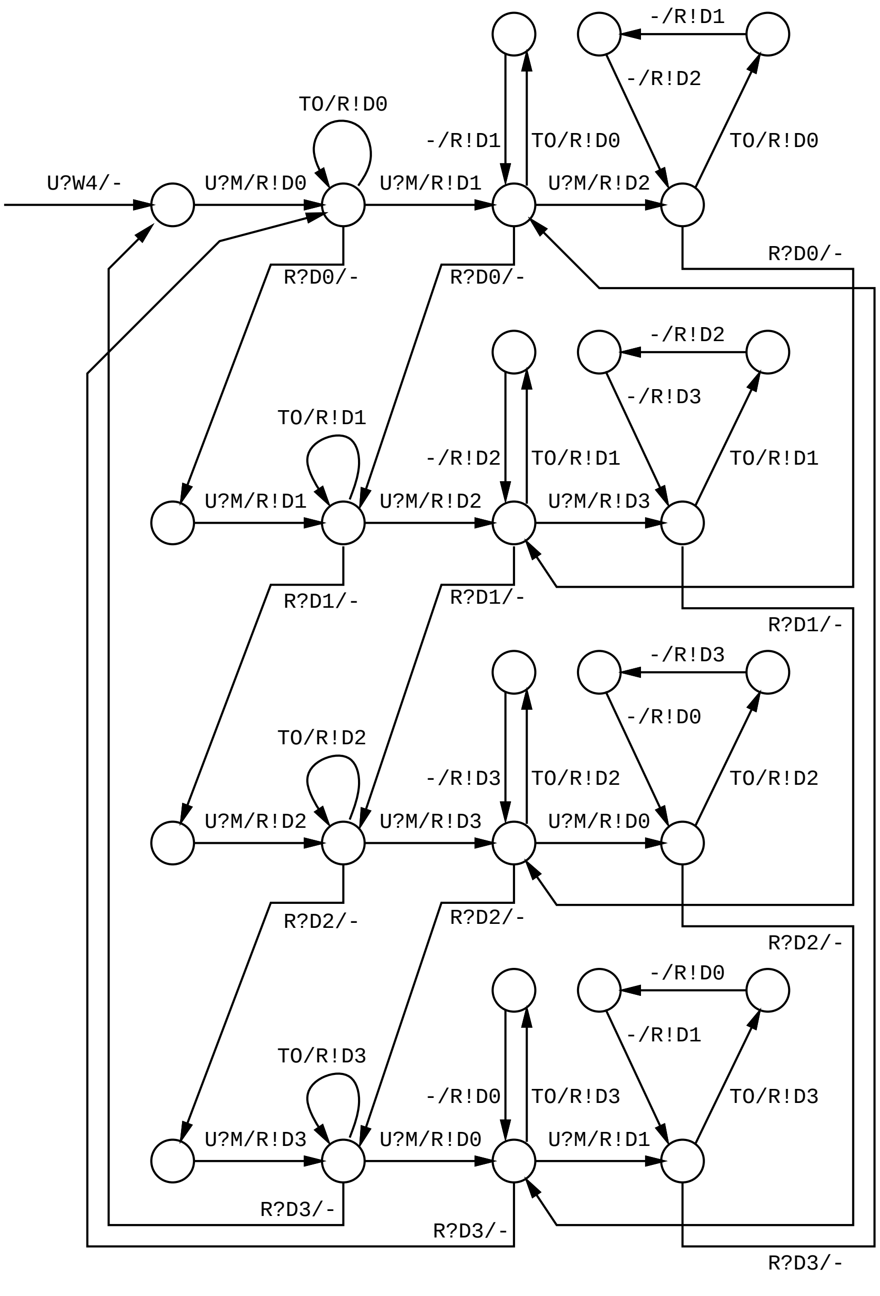


Figure 17: An FSM specification for Go-Back-N Protocol with window size 4

of the specification. While testing the control part is relatively easy, testing the data part is not. Worse, when data and control interfere with each other, testing becomes even more difficult. Therefore, TCG for an extended model seems to be an "untouchable" problem in the past few years. In this chapter, some of the pioneer research appeared in the literature is introduced.

4.2 Conformance Testing for EFSM and Estelle

4.2.1 A Brief Introduction of Estelle Syntax

Estelle [7, 13, 26] is a description language for EFSM. Similar to the EFSM model, a transition in EFSM points from its head state to its tail state. The head and tail states are defined in the **from** and **to** clauses, respectively. The actions of a transition can be any Pascal statement sequence[1] quoted by reserved words **begin** and **end**. The action can be executed only when the input specified in the **when** clause is given, and the predicate in the **provided** clause is satisfied.

A sample transition is shown in Figure 18. The transition starts from state s_h and goes to state s_t (the **from** and **to** clauses) only when an input message m is given through *Interaction Points (IP)* p (the **when** clause) and the boolean expression $\mathcal{C}$ is satisfied (the **provided** clause). The variables x_1, x_2, ..., x_n are the arguments of message m. They will be assigned the corresponding values carried by message m. The action of the transition is a sequence of statements listed between **begin** and **end**. In Figure 19, The transition in Figure 18 is redrawn using the EFSM model described in the previous section. The **when** clause act like an input action and the

[1]There are some restrictions and extensions imposed on the Pascal syntax. Reference [26] for detail.

provided clause act like an conditional actions. However, the difference is that when the predicate in the **provided** clause is not satisfied, the **when** clause will not be taken.

from s_h
to s_t
when $p.m(x_1, x_2, ..., x_n)$
provided $\mathcal{C}$
  begin S_1; S_2; ... S_m **end;**

Figure 18: A sample Estelle transition

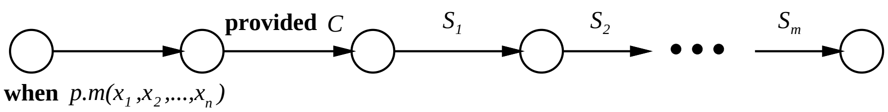


Figure 19: The sample Estelle transition in Figure 18 shown as a sequence of EFSM transitions

A *Normal Form Specification (NFS)* of an Estelle specification is that every transition is specified in the format shown in Figure 18. The transition in the NFS is called a *Normal Form Transition (NFT)*.

An output statement in Estelle is written as

$$\textbf{output}\ p.m(e_1, e_2, ..., e_n)$$

where p is an IP, m is the output message, and e_1, e_2, ..., e_n are expressions. The result of the expressions will be sent alone with message m to another IP that receives the message.

Estelle allows multiple EFSM defined in a specification. Each EFSM is called a *module*, and can communicate with other modules through *channels*. A channel is defined as follows.

$$
\begin{array}{l}
\textbf{channel } ch(role_1, role_2); \\
\quad \textbf{by } role_1: \\
\quad\quad M_1;\ M_2;\ ...;\ M_k; \\
\quad \textbf{by } role_2: \\
\quad\quad M_{k+1};\ M_{k+2};\ ...;\ M_l; \\
\quad \textbf{by } role_1,\ role_2: \\
\quad\quad M_{l+1};\ M_{l+2};\ ...;\ M_n;
\end{array}
$$

Each M_i above is in the form of

$$msg_i(x_{i1} : t_{i1}; x_{i2} : t_{i2}; ...; x_{im_1} : t_{im_1})$$

where msg_i is the name of the message and the x's are the parameters of the message and t's are the types of the corresponding parameters. The channel specification shown above declares a channel ch with its two end points playing $role_1$ and $role_2$. Messages M_1 to M_k is able to be sent by $role_1$, while messages M_{k+1} to M_l can be sent by $role_2$. Messages M_{l+1} to M_n can be sent by both. The messages that can be sent by $role_1$ can be received by $role_2$, and vice versa.

An IP is declared after the reserved word, **ip**. The format is as follows.

$$
\begin{array}{ll}
\textbf{ip} & p_1 : ch_{c_1}(role_{r_1}); \\
 & p_2 : ch_{c_2}(role_{r_2}); \\
 & \quad\quad \vdots \\
 & p_n : ch_{c_n}(role_{r_n});
\end{array}
$$

where ch_{c_i} is the name of the channel that p_i will be connected, and $role_{r_i}$ is the role that p_i will play. The IPs are connected by the **connect** and **attach** statements and are disconnected using the **disconnect** and **detach** statements. These statements have the following format.

$$\left\{\begin{array}{l}\textbf{connect}\\ \textbf{disconnect}\\ \textbf{attach}\\ \textbf{detach}\end{array}\right\} m_1.p_1 \textbf{ to } m_2.p_2$$

where m_1 and m_2 are module names (described below), and p_1 and p_2 are IP names.

The outlook of a module is declared in the **module** declaration, and the interior is defined in the **body** definition. The former is called a *module type* and the latter is called a *module body*. A module type defines the IPs and *exported variables* that can be seen by other modules, while a module body specifies local IPs, variables, and the transitions of the EFSM. Note that Estelle allows nested modules; i.e., modules can be defined locally within another module body. A *module variable* is a realization of a module, and is declared in the **modvar** clause. A module variable is declared to be a module type in the **modvar** clause and is then associated with a module body using the **init** clause.

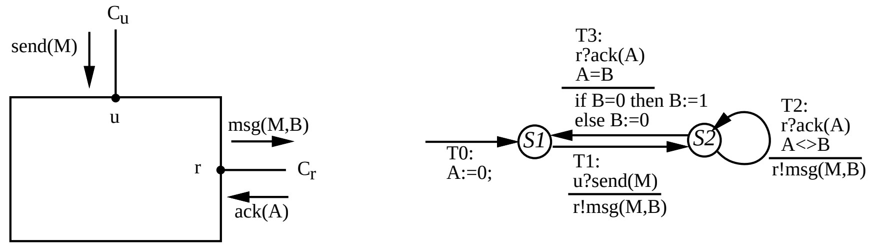
Figure 20 shows the relationship among the module types, module bodies, and module variables. In the figure, a module type $M1$, and two module bodies $B1$ and $B2$ are declared. Since $B1$ and $B2$ are declared to be in $M1$ type, their appearance will be the same; both can be accessed through IPs p and q, for example. Module variables $V1$ and $V2$ are declared to be $M1$ and are associated to $B1$ and $B2$, respectively, using the **init** statements. That is, although $V1$ and $V2$ look similar from the outside, their behavior may differ because of their different module bodies. The **connect** clause in the figure shows that both $V1$ and $V2$ contains its own IPs p and q, and IP p of $V1$ is connected to IP q of $V2$.

There are other declarations, statements and clauses that have not introduced here. For detailed syntax of Estelle, please reference to [26].

The same example of the ABP's sender in the previous sections is shown in Figures 21 and 22. Figures 21a and 21b show the structure and the state diagram of the protocol, respectively. There are two interaction points for the ABP sender. One is connected to channel c_u, which takes sending request from the upper layer protocols ($send(M)$). The other is connected to channel c_r, through which messages ($msg(M, B)$) and acknowledgments ($ack(A)$) are sent and received, respectively. The transitions in the EFSM are labeled from T_0 to T_3. Transition T_0 is the *initial transition*, which sets variable b to zero. Upon receiving a message $send(M)$ from IP u, the EFSM sends a message $msg(M, b)$ through IP r to the receiver (transition T_1). The message contains the data (M) and a sequence number for the message (B). If an acknowledgment $ack(a)$ is received from IP r and the sequence number A is not equal to B, errors must have occurred and the message has to be resent again (transition T_2). If the received acknowledgment has the same sequential number as the message that has just been sent, the EFSM changes the value of B and becomes ready for transmitting the next message. Each transition in Figure 21b is translated into Estelle in Figure 22.

```
channel ch(role1,role2);
    by role1:
        msg1;
    by role2:
        msg2;

module M1;
    ip p:ch(role1);
       q:ch(role2);
    ....
end;

body B1 of M1;
    ....
end;

body B2 of M1;
    ....
end;

modvar
    V1: M1;
    V2: M1;

initialize
    begin
        ....
        init V1 with B1;
        init V2 with B2;
        connect V1.p to V2.q;
        ....
    end;
```

Figure 20: Declaration of module types, module bodies, and module variables



(a) (b)

Figure 21: The sender of ABP specified in Estelle (The module structure and the state diagram)

```
initialize   from S1                from S2               from S2
to S1        to S2                  to S2                 to S1
name T0      when u.send(M)         when r.ack(A)         when r.ack(A)
    begin    name T1                provided A<>B         provided A=B
    B:=0         begin              name T2               name T3
    end;         output r.msg(M,B)      begin                 begin
                 end;                   output r.msg(M,B)     if b=0 then B:=1
                                        end;                  else B:=0
                                                              end;
```

Figure 22: The sender of ABP specified in Estelle (Transitions specified in NFS)

4.2.2 Conformance Testing

Almost every TCG method proposed for EFSM or Estelle so far separates control analysis from data analysis. The methods described in this section [9, 53, 50], basically follow these steps:

1. Derive control flow graph. Use the existing TCG methods for FSM to analyze the graph.

2. Derive a data flow graph. List all dependencies of the variables.

3. Select a path to be tested. Identify the path between the transitions where a variable is defined and the transition where the same variable is used. Make sure that the value of the variable used is consistent with the value it defined.

4. Find a test case that will detect errors in the selected path.

The steps are described in the following sections. The methods proposed by Ural and Sarikaya are based on Estelle, while the one proposed by Chanson is based on EFSM. Since the underlying structure of an Estelle specification is an EFSM, Estelle and EFSM will be used interchangeably in this section.

4.2.3 Control Flow Analysis

Deriving a control flow graph is quite straight forward. In both [9] and [50], the action associated with each transition in an EFSM is replaced with a label. Doing so transforms an EFSM to an FSM. Applying the TCG methods described in the previous chapter, the test case for control flow can be derived.

Ural and Yang, however, defines a control graph in more detail [53]. They define s-nodes for the states, i-nodes for the input actions, and t-nodes for the transitions. An s-node has an si-edge pointing to an i-node if its corresponding state has an outgoing transition containing a **when** clause. An it-edge points from an i-node to a t-node if the t-node is the action (the **begin-end** clause) follows the i-node. If there is any predicate (**provided** clause), the predicate is label beside the it-node. An s-node can also points to a t-node through an st-edge if there are no **when** clause between them. The predicate, if any, is also labeled beside the st-edge. Finally, a ts-edge points from a t-node to an s-node if state s is contained in the **to** clause.

As Chanson pointing out in [9], a test case found by the FSM's TCG methods may not be generated by a legal path; i.e., the corresponding transition sequence may not be executable. Recall that some transition in Estelle or EFSM can be executed only when certain conditions hold. Since data dependencies are involved, the execution of a transition may prevent another transition to be executed. However, none of the authors states how to resolve this problem. Beside, it is still unclear what data should be applied to the implementation in order to drive the EFSM trough the path. It is as well unclear what should be observed from the implementation in order to determine the correctness of the control.

4.2.4 Data Flow Analysis

The methods discussed in this chapter differ from those in Chapter III in their data flow analysis. Sarikaya defines four types of nodes: the I-nodes, representing input primitive parameters, the D-nodes, representing context variables and constants, the

O-nodes, representing the output parameters, and the F-nodes, repressing data operations. The actions of the transitions can all be simplified as assignment operations (see Section 2.3). Edges of the data flow graph is drawn from the nodes representing the elements at the right hand side of an assignment statement (D-, F-, or I-nodes) to the node representing the left hand side of the statement (D- or O-nodes).

Ural and Yang do not define a data graph explicitly. Instead, they label the defines and uses of the variables beside the nodes. If a variable is set to a new value, an entry indicating the variable being used is listed beside the i- and t-nodes. If a value is referenced by a statement in the i- and t-nodes, an entry describing their use is also listed.

Chanson and Zhu provides a procedure that exhaustively search for all *def-clear paths.* A def-clear path for a variable v is a path in which v is defined in the first transition and is firstly used in the last transition of the path; i.e., no other transition in the path uses the variable.

4.2.5 Path Selection

With data analysis provided, paths can be chosen to observe the correctness of the data. Different path selection strategies can be used and some of the frequently referenced strategies are define as follows:

Path testing: Every executable path must be covered by a test case [24]. This approach is impractical since even a small program may derive huge or infinite number of paths.

Branch testing: Each branch of control must be tested by a test case [24]. This is usually considered the minimum test requirement.

Segment (statement) testing: Each statement of a program must be executed by at least one test case [24].

Boundary-interior path testing: Loops in a program are only tested twice: one without iterating, and the other iterate at least once [22].

Structured path testing: Similar to boundary-interior path testing, except that the structured path testing iterates at least k times, instead of once [23].

In [47], Rapps and Weyuker proposes another set of path selection policies. They defines the usage of variable as computation (*c-use*, such as the right hand side of an assignment statement) and predicate (*p-use*, such as the condition of a **provided** clause or an **if** statement). In the following description of the policies, assume that a variable x is defined at a transition t.

All-uses: Every def-clear path w.r.t. x from t that leads to a c- or p-use of x must be tested.

All-defs: Some def-clear paths w.r.t. x from t that lead to a c- or p-use of x are tested.

All-p-uses: Every def-clear path w.r.t. x from t that leads to a p use must be tested.

All-c-uses/some-p-uses: Every def-clear path w.r.t. x from t that leads to a c-use of x, or some def-clear paths w.r.t. x from t that lead to a p-use are tested.

All-p-uses/some-c-uses: Every def-clear path w.r.t. x from t that leads to a p-use of x, or some def-clear paths w.r.t. x from t that lead to a c-use are tested.

All-du-paths: Every def-clear loop-free path w.r.t. x from a transition that defines x to a transition that uses x must be tested.

Other strategies includes *2-dr interaction* [43], *required pairs strategy* [42], *elementary data context* [36], *ordered data contexts* [36], *required k-tuples* [43], and *definition-tree testing* [35]. These strategies are ordered with respect to their fault coverage in Figure 23, in which the higher the position, the wider the fault coverage is [44].

With the data flow graph generated in the previous steps, Sarikaya, Ural, Chanson and their colleagues are able to select necessary path to test. The path selection policies used in their papers are the all-du-path strategy. What they do is to find a variable, locate its definition and usages, and search for all def-clear path between the definition and the usages. The I/O event that drives the EFSM through the path are used as test cases.

4.2.6 Test Case Selection

Having selected the control and data paths to test, one still need to determine what data is needed to drive an EFSM through the desired paths. Unfortunately, none of the methods mentioned above consider test case selection. The test cases are derived *ad hoc* by manually tracing through the paths.

It is not so easy to derive a test case for a certain path. As mentioned before, control and data flows often interfere with each other. One cannot direct the control

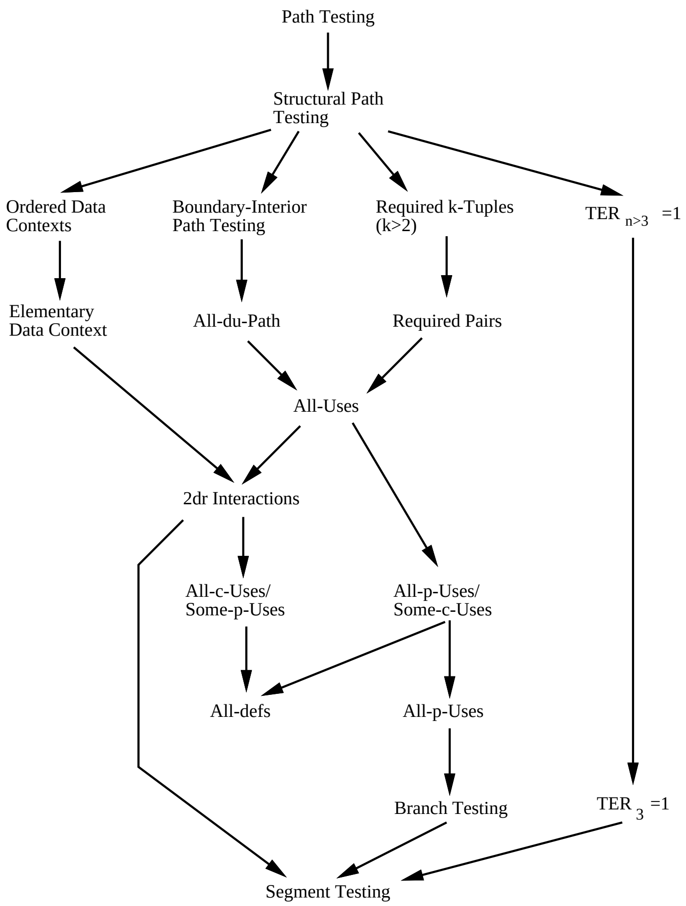


Figure 23: Partial ordering of path selection strategies

flow without providing some needed data, and neither can one test the data without considering how the control flow goes. Moreover, it is proven that in an extended model, whether a path can be traversed is undecidable in the most general cases [39]. Therefore, it is hard for human being to trace through every possible combination to find a test case; automatic test case generation is desperately needed.

4.3 Conformance Testing for LOTOS

4.3.1 A Brief Introduction of LOTOS Syntax

LOTOS [4, 27] describes a system as a hierarchy of *processes*. A process is an entity able to perform internal, unobservable actions, and to interact with other processes. A process communicates with its outside world through *gates*. There is a simplified version of LOTOS, called *Basic LOTOS*, which omits the data structures from the *Full LOTOS*, and specified only the behavior and the interaction among processes. The full LOTOS, on the other hand, is able to define data types and variables. The data can be passed alone with messages to other processes; just as it is in Estelle. Since the TCG methods that will be described in Section 4.3 are all deal with basic LOTOS, full LOTOS will not be discussed here. Readers who are interested in full LOTOS is referred to [4, 27].

Figures 24 and 25 specifies a sample LOTOS process borrowed from [4]. Process `Max3` is composed of two `Max2` processes. The former has four gates, while the latter has three (Figure 24). The syntax of basic LOTOS is relatively simple compared to Estelle's. A *process definition* line includes the process name and a list of gates that serves as the interface between the process and its outside world. The process definition is followed by a *behavior expressions* which describe how the process functions. Subprocesses used in the behavior expressions are defined after the keyword **where** in the same manner as their super-processes.

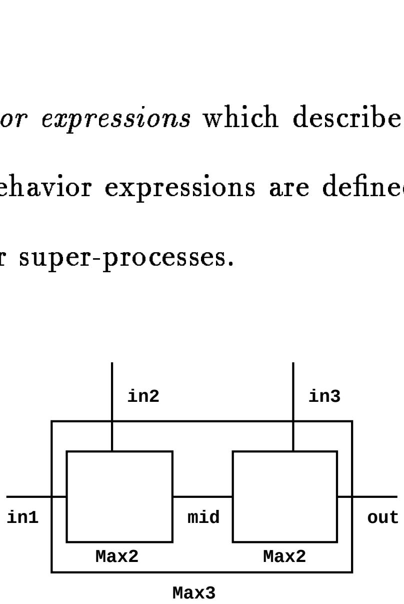


Figure 24: A simple LOTOS process (The process structure)

```
process Max3[in2, in2, in3, out] :=

  hide mid in
      (Max2[in1,in2,mid] |[mid]| Max2[mid,in3,out])

  where
       process Max[a, b, c] :=

           a; b; c; stop
           []
           b; a; c; stop

       endproc
endproc
```

Process Definition

Behavior Expression

Definition of Subprocesses

Behavior Expression

Figure 25: A simple LOTOS process (A LOTOS specification)

Figure 25 shows that process `Max3` behaves like two `Max2` processes synchronized by an internal gate "`mid`." Each `Max2` process takes actions on the three gates in the order of either a, b, and c, or b, a, and c. Note that LOTOS does not differentiate input actions from the output ones. The actions on the gates merely specified the occurrence of an event. These events can be synchronized if specified. For example, the operator "`|[mid]|`" synchronizes the third gate of the first `Max2` and the first gate of the second `Max2`. If any of the `Max2` process has not create a "`mid`" action, the other must wait. The behavior of a process can be described as an *action tree*, which shows possible sequence of actions for the process. The action trees of the two `Max2` processes are shown in Figures 26a and 26b, and the action tree of `Max3`, which is constructed from the previous two, is shown in Figure 26c.

In addition to the operators, ";", "[]", and "`|[mid]|`," shown in Figure 25, there are other operators that can be used to specify a behavior expression. The syntax of behavior expressions are described as interaction of processes. For example, the behavior expression of `Max2` can be described as $B_1[]B_2$, where B_1 behaves like "a; b; c; **stop**" and B_2 behaves like "b; a; c; **stop**." Furthermore, "a; b; c; **stop**" can be represented as "$a; B_3$," where B_3 is "b; c; **stop**." The B_1, B_2, and B_3 above are all processes. In other words, the operators in a behavior expression describe the interaction among processes. The operators of LOTOS is described as follows, in which the statements quoted in a pair of parentheses denotes the described operations.

Inaction (stop): Specifies a process that does nothing.

Action prefix:

Unobservable (i; B): An unobservable action takes place before process B starts.

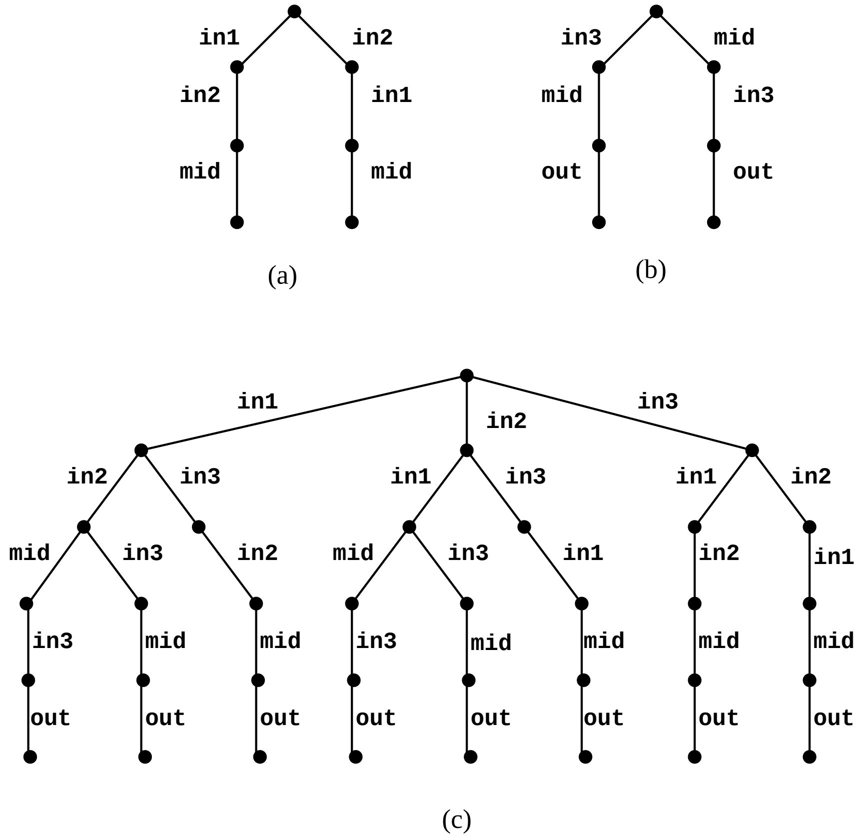


Figure 26: Action trees for the processes in Figure 25

Observable ($g; B$): An observable action at gate g occurs before process B starts.

Parallel Composition:

General Case ($B_1 \,|[g_1, ..., g_n]|\, B_2$): Processes B_1 and B_2 run concurrently, but are synchronized by gates g_1 to g_n; i.e., if B_1 makes an action on gate g_i, it must wait for B_2 to act on g_i before it can proceed (or vice versa), where i is between 1 and n. Gates g_1 to g_n are called synchronization gates.

Pure Interleaving ($B_1 \,|||\, B_2$): Processes B_1 and B_2 run in parallel. That is, the set of synchronization gates between B_1 and B_2 is empty.

Full Synchronization ($B_1 \,||\, B_2$): The set of synchronization gates between B_1 and B_2 contains all gates. Therefore, B_1 and B_2 lock up each other in every steps.

Hiding (hide $g_1, ..., g_n$ in B): Declares the actions on gates g_1 to g_n to be internal actions. (Thus, a more accurate action tree for `Max3` is replacing each `mid` in Figure 26 with **i**.)

Process Instantiation ($p[g_1, ..., g_n]$): This process is a process declared after keyword **where**, or is a process that has been declared before.

Successful Termination (exit): The process terminated successfully and takes no more actions.

Sequential Composition ($B_1 >> B_2$): If B_1 terminated successfully, process B_2 is enabled.

Disabling ($B_1[> B_2$): Process B_1 can be disabled by B_2; i.e., once B_2 starts executing, B_1 stopped. If B_1 terminates successfully before B_2 interrupts it, B_2 terminates as well.

An example of the sender of the ABP is shown in Figure 27. Process `Sender` has five gates: `M` to receive a sending request from its upper layer users, `D0` and `D1` to send data with sequence numbers 0 and 1, respectively, and `A0` and `A1` to receive acknowledgements from the receiver. Subprocess `Send` sends data through gate `Data`, and then receives an acknowledgment from either gate `GoodAck` or gate `BadAck`. If the former is received, `Send` terminates successfully. Otherwise, it recursively calls itself to send the same data again. Process `Sender` calls `Send` after receiving a request from gate `M`. The data is sent through `D0` and the correct and incorrect acknowledgments are sent through `A0` and `A1`, respectively. After `(M; Send[D0,A0,A1])` successfully terminated (**exit** called in subprocess `Send`), data $D1$ can be sent in the same manner as $D0$. The process recursively calls itself in order to send more messages.

4.3.2 Operational Semantics of LOTOS

The *operational semantics* (also called *labeled transition systems*) of LOTOS provides a mean to systematically derive the actions from a process (or a behavior expression). An informal presentation of the semantics of a process has been shown as the action tree in Figure 26. The same trees can be represented in a mathematical manner with operational semantics [4, 6, 27].

```
process Sender[M,D0,A0,D1,A1] :=

        (M; Send[D0,A0,A1]) >> (M; Send[D1,A1,A0]) >> Sender[M,D0,D1,A0,A1]

    where
        process Send[Data,GoodAck,BadAck] :=

                Data;
                (GoodAck; exit
                 []
                 BadAck; Send(Data,GoodAck,BadAck)
                )
        endproc
endporc
```

Figure 27: LOTOS specification of ABP

The elements in the operational semantics of LOTOS follows these conventions.

1. L denotes the set of observable events;

2. $\mu \in L$ is an observable event;

3. L^* represents a set of observable events sequence;

4. $\sigma \in L^*$ is an observable event sequence;

5. τ denotes the unobservable (internal) action (generated by **i**);

6. δ is the successful termination action (generated by **exit**);

7. $\alpha \in L \cup \{\tau\}$

8. $\alpha^+ \in L \cup \{\tau, \delta\}$

9. B or C denotes a process (or a behavior expression).

The notation, $B_1 -\mu\rightarrow B_2$, in a labeled transition system denotes process B_1 changes to process B_2 when an action μ occurs. Other notations are defined in Table 7, and the formal definition of the operators in Section 4.3.1 are shown in Table 8. Note that, $Tr(B)$ is called the *trace* of process B.

Table 7: Notation of the operational semantics for LOTOS

Notation	Meaning and Explanation
$B -\mu_1...\mu_n\rightarrow C$	$\exists B_i(1 \leq i \leq n)\ [B = B_0 -\mu_1\rightarrow B_1 -\mu_2\rightarrow \ ...\ -\mu_n\rightarrow B_n = C]$ Process B changes to C after a sequence of actions $\mu_1, ..., \mu_n$.
$B =\tau\Rightarrow C$	$(B \equiv C) \vee (\exists n \geq 1\ [B -\tau^n\rightarrow C])$ Process B changes to C without any action is either (1) B and C are the same process, or (2) B changes to C through a sequence of unobservable actions.
$B =\mu\Rightarrow C$	$\exists B_1, B_2[B =\tau\Rightarrow B_1 -\mu\rightarrow B_2 =\tau\Rightarrow C]$ Process B changes to C through the action μ and possibly some internal actions.
$B =\mu_1...\mu_n\Rightarrow C$	$\exists B_i(1 \leq i \leq n)\ [B = B_0 =\mu_1\Rightarrow B_1 =\mu_2\Rightarrow \ ...\ =\mu_n\Rightarrow B_n = C]$ Process B changes to C after a sequence of external actions $\mu_1, ..., \mu_n$ and some internal actions.
$B =\mu_1...\mu_n\Rightarrow$	$\exists C\ [B =\mu_1...\mu_n\Rightarrow C]$ Action sequence $\mu_1...\mu_n$ can occur in process B.
$B \neq\mu_1...\mu_n\Rightarrow$	$\neg\exists C\ [B =\mu_1...\mu_n\Rightarrow C]$ Action sequence $\mu_1...\mu_n$ cannot occur in process B.
$out(B)$	$\{\mu \in L \mid B =\mu\Rightarrow\}$ The set of all possibly immediate actions of process B
$Tr(B)$	$\{\sigma \in L^* \mid B =\sigma\Rightarrow\}$ The set of all possibly action sequences of process B

Table 8: Semantics of the LOTOS operators

Operator	Semantics and Explanation
stop	$\forall \mu\ [\textbf{stop} \not\!\!-\!\mu\!\!\Rightarrow]$ A stopped process takes no actions.
$\alpha; B$	$\mu; B \ -\!\mu\!\!\rightarrow B \qquad \textbf{i}; B \ -\!\tau\!\!\rightarrow B$ An action μ creates a process B. An internal action directly change the process to B without any observable actions.
$B_1[]B_2$	$B_1 \ -\!\alpha^+\!\!\rightarrow B_1' \implies B_1[]B_2 \ -\!\alpha^+\!\!\rightarrow B_1'$ $B_2 \ -\!\alpha^+\!\!\rightarrow B_2' \implies B_1[]B_2 \ -\!\alpha^+\!\!\rightarrow B_2'$ If B_1 starts first, executes B_1. Otherwise, executes B_2.
$B_1 \ \|[g_1, ..., g_n]\| \ B_2$	Let $S = \{g_1, ..., g_n\}$ $B_1 \ -\!\alpha\!\!\rightarrow B_1' \wedge \alpha \notin S \implies B_1 \ \|[S]\| \ B_2 \ -\!\alpha\!\!\rightarrow B_1' \ \|[S]\| \ B_2$ $B_2 \ -\!\alpha\!\!\rightarrow B_2' \wedge \alpha \notin S \implies B_1 \ \|[S]\| \ B_2 \ -\!\alpha\!\!\rightarrow B_1 \ \|[S]\| \ B_2'$ $(B_1 \ -\!\alpha^+\!\!\rightarrow B_1') \wedge (B_2 \ -\!\alpha^+\!\!\rightarrow B_2') \wedge (\alpha^+ \in S \cup \{\delta\})$ $\implies B_1 \ \|[S]\| \ B_2 \ -\!\alpha^+\!\!\rightarrow B_1' \ \|[S]\| \ B_2'$ Gate α^+ in $S \cup \{\delta\}$ synchronizes B_1 and B_2. Other actions make B_1 or B_2 move freely.
$B_1 \ \|\|\| \ B_2$	Same as above, except $S = \emptyset$ Since there is no gate to synchronize B_1 and B_2, they are executed concurrently.
$B_1 \ \|\| \ B_2$	Same as above, except $S = G$ Since B_1 and B_2 are synchronized by all gates, they lock up each other on every step.
exit	$\textbf{exit} \ -\!\delta\!\!\rightarrow \textbf{stop}$ The process terminates successfully (action δ), and stops.
$B_1 >> B_2$	$B_1 \ -\!\alpha\!\!\rightarrow B_1' \implies B_1 >> B_2 \ -\!\alpha\!\!\rightarrow B_1' >> B_2$ $B_1 \ -\!\delta\!\!\rightarrow B_1' \implies B_1 >> B_2 \ -\!\tau\!\!\rightarrow B_2$ If B_1 is still executing, B_2 cannot start. If B_1 terminates, B_2 is started immediately.
$B_1[> B_2$	$B_1 \ -\!\alpha\!\!\rightarrow B_1' \implies B_1[> B_2 \ -\!\alpha\!\!\rightarrow B_1'[> B_2$ $B_1 \ -\!\delta\!\!\rightarrow B_1' \implies B_1[> B_2 \ -\!\delta\!\!\rightarrow B_1'$ $B_2 \ -\!\alpha^+\!\!\rightarrow B_2' \implies B_1[> B_2 \ -\!\alpha^+\!\!\rightarrow B_2'$ Normally, B_1 can run freely. If B_1 terminates successfully, the whole process terminated successfully. However, once B_2 stars execution, B_1 stops and B_2 takes over.

4.3.3 Formal Definition of Conformance Testing

Brinksma proposes a formal definition of conformance testing in terms of LOTOS notations [5, 6]. Such definition yields an elegant theory about conformance testing, which will be discussed in Sections 4.3.4 and 4.3.5.

The terminology of conformance testing described in Section 2.1 can be formally defined in LOTOS notations as follows.

Definition 4.3.1

1. *A* test suite TS *is a set of processes. The elements of a test suite are called* test cases.
2. *Let* $\sigma \in L^*$, T *be a test case and* B *be a process. Then, a derivation* $T \mid\mid B = \sigma \Rightarrow T' \mid\mid B'$ *is a* test run *of* T *and* B. *A test run is* complete *if* $T \mid\mid B = \sigma \Rightarrow$ **stop**.
3. *A complete test run* $T \mid\mid B = \sigma\delta \Rightarrow$ **stop** *is called a* successful *test run. A complete test run is* failed *if it is not successful.*

Since a test case T is defined as a process, it can be derived into a set of action sequences. Let B be the behavior of an implementation. Any action sequences that exists in T must also exist in B if the implementation conforms to the specification. Therefore, when processes T and B are executed in completely synchronize (i.e., $T \mid\mid R$; taking exactly the same action in every step), one of them should be able to terminated successfully. Note that such defined test case T can only used to ensure the weak conformance.

4.3.4 Relationships between Processes

First, Brinksma defines that a LOTOS process is a *reduction* of another if both process follows Definition4.3.2.

Definition 4.3.2 *Let B_1 and B_2 be two processes.* B_1 **red** B_2 *iff*

1. $Tr(B_1) \subseteq Tr(B_2)$, *and*

2. $\forall \sigma \in L^*$, $\forall A \subseteq L$,

   if $\exists B_1', \forall a \in A, [B_1 =\sigma\Rightarrow B_1' \not= a\Rightarrow]$

   then $\exists B_2', \forall a \in A, [B_2 =\sigma\Rightarrow B_2' \not= a\Rightarrow]$

That is, process B_1 is a reduction of B_2 if (1) all observable action sequences that can occur in B_1 must be able to occur in B_2, and (2) an observable action sequence that causes B_1 to stop, will cause B_2 to stop as well. For example, in Figure 28, processes B and F are reductions of process A. (The **i**'s in the figure represent internal actions.) On the other hand, processes C and E are not reductions of A because $\langle ab \rangle \in Tr(C)$, but $\langle ab \rangle \notin Tr(A)$ (i.e., $Tr(C) \not\subseteq Tr(A)$). Process D is not a reduction of A either, for it stops when applied action sequence $\langle b \rangle$.

Similarly, the *extension* relationship between processes is defined as follows.

Definition 4.3.3 *Let B_1 and B_2 be two processes.* B_1 **ext** B_2 *iff*

1. $Tr(B_1) \supseteq Tr(B_2)$, *and*

2. $\forall \sigma \in Tr(B_2)$, $\forall A \subseteq L$,

   if $\exists B_1', \forall a \in A, [B_1 =\sigma\Rightarrow B_1' \not= a\Rightarrow]$

   then $\exists B_2', \forall a \in A, [B_2 =\sigma\Rightarrow B_2' \not= a\Rightarrow]$

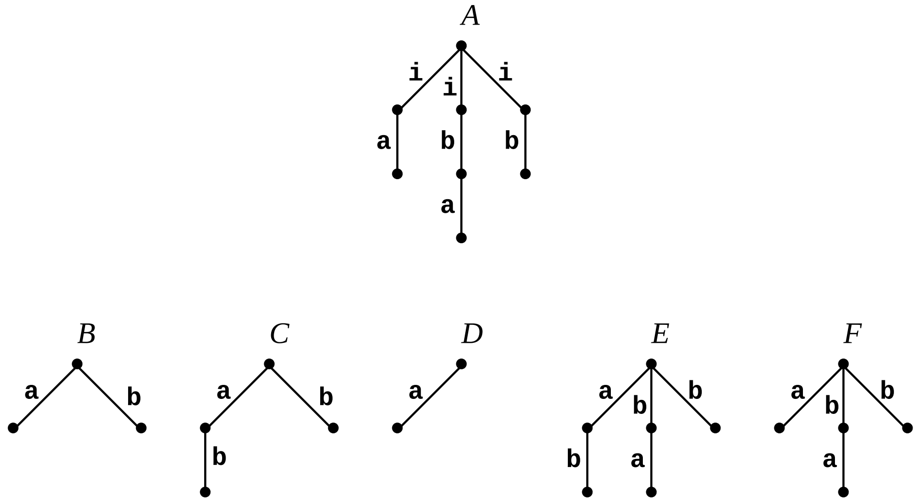


Figure 28: Sample processes illustrating relationships **red**, **ext**, and **conf**

That is, process B_1 is an extension of B_2 if (1) B_1 can execute every possible trace of B_2, and (2) for those traces of B_2 that makes B_1 stop, they also make B_2 stop. For example, in Figure 28, processes E and F are both extensions of process A.

One process *conforms* to another if the two processes satisfies Definition 4.3.4.

Definition 4.3.4 *Let B_1 and B_2 be two processes. B_1* **conf** *B_2 iff*

$$\forall \sigma \in Tr(B_2),\ \forall A \subseteq L,$$

$$\textit{if } \exists B_1', \forall a \in A, [B_1 = \sigma \Rightarrow B_1' \not= a \Rightarrow]$$

$$\textit{then } \exists B_2', \forall a \in A, [B_2 = \sigma \Rightarrow B_2' \not= a \Rightarrow]$$

In other words, the conformance relation is the union of both reduction and extension. For example, processes B, E, and F are all conform to A.

Similarly, the intersection of reduction and extension relationships is called *testing equivalence*, and is denoted by a "$\sim$."

Definition 4.3.5 *Let B_1 and B_2 be two processes. $B_1 \sim B_2$ iff*

$$(B_1 \textbf{ red } B_2) \wedge (B_1 \textbf{ ext } B_2) \tag{4.1}$$

4.3.5 Canonical Tester

Let T be a test case and B be a process, $Succ(T, B)$ denotes that all complete test run of T and B are successful. That is

$$\forall \sigma \in L^*, T \mid\mid B = \sigma \Rightarrow \textbf{stop} \tag{4.2}$$

Similarly, $Succ(TS, B)$ for a test suite TS means $Succ(T, B)$ for all T in TS. Then, a *canonical tester* is defined as follows.

Definition 4.3.6 *Let S be a process. The canonical tester $T(S)$ is a process such that:*

1. $Tr(T(S)) = Tr(S)$
2. $\forall B, B \textbf{ conf } S, Succ(T(S), B)$.

In other words, a canonical tester is able to run successfully on any implementation B that is conform to the specification S.

Let B be a process. A *refusal set* $R(B')$ of process B' is defined as follows.

Definition 4.3.7

$$R(B') = \{A \in 2^L \mid \exists\sigma \in Tr(B), \forall\mu \in A[B =\sigma\Rightarrow B' \not=\mu\Rightarrow]\} \tag{4.3}$$

where 2^L *represent the power set of* L.

That is, if B' can be reached from B by some sequence of actions, $R(B')$ contains those action sets whose elements cannot be accepted by process B'. A *failure tree* $F(B)$ is an action tree of process B with each note s associated with its refusal set $R(s)$. It can be proved that $B_1 \sim B_2$ iff $F(B_1) = F(B_2)$.

As mentioned in Definition 4.3.6, for any process B that conforms to S, $T(S) \mid\mid B =\sigma\Rightarrow$ **stop**. Since S conforms to itself, same thing must hold for process S; i.e, $T(S) \mid\mid S =\sigma\Rightarrow T(S)' \mid\mid S' \sim$ **stop**. Because of $T(S)' \mid\mid S' \sim$ **stop**, process $T(S)' \mid\mid S'$ refuses everything; i.e., $L \in R(T(S)' \mid\mid S')$. It can be proved that if $A_1 \in R(B_1)$ and $A_2 \in R(B_2)$, then $A_1 \cup A_2 \in R(B_1 \mid\mid B_2)$. Therefore, $T(S)$ must be so constructed that its final refusal set complements the one of process S. Thus, $T(S)$ can be constructed by the following steps.

1. Construct $F(S)$.
2. Construct $F(S)'$, whose underlying action tree is the same as $F(S)$ and the refusal $R(s)'$ for each node s follows these rules:
   (a) $R(S)' = \{L - A \mid A \notin R(s), A \subseteq L\}$ if $L \notin R(s)$
   (b) $R(S)' = 2^L$ if $L \in R(s)$
3. Construct a process $T(S)$ such that $F(T(S)) = F(S)'$.

Assume that B does not conform to S. According to Definition 4.3.4, there exists a sequence of action $\sigma \in L^*$ and an action a such that, $B = \sigma \Rightarrow B' \neq a \Rightarrow$ but $S = \sigma \Rightarrow S' = a \Rightarrow$. That is, $a \in R(B')$ but $a \notin R(S')$. Step 2 above adds the complement of $R(B')$ to $F(T(S))$, which guarantees $T(S)$ and B' deadlocks if B does not conform to S.

Figure 29 is an example of how $T(S)$ can be constructed. The failure tree of Figure 29a is shown in Figure 29b. Applying step 2 above, Figure 29c is derived, and a test case $T(S)$ is found in Figure 29d. Similarly, the $T(S)$ for Figure 29d is derived through Figures 29e, f, and a. Figure 29a and 29d are canonical testers for each other.

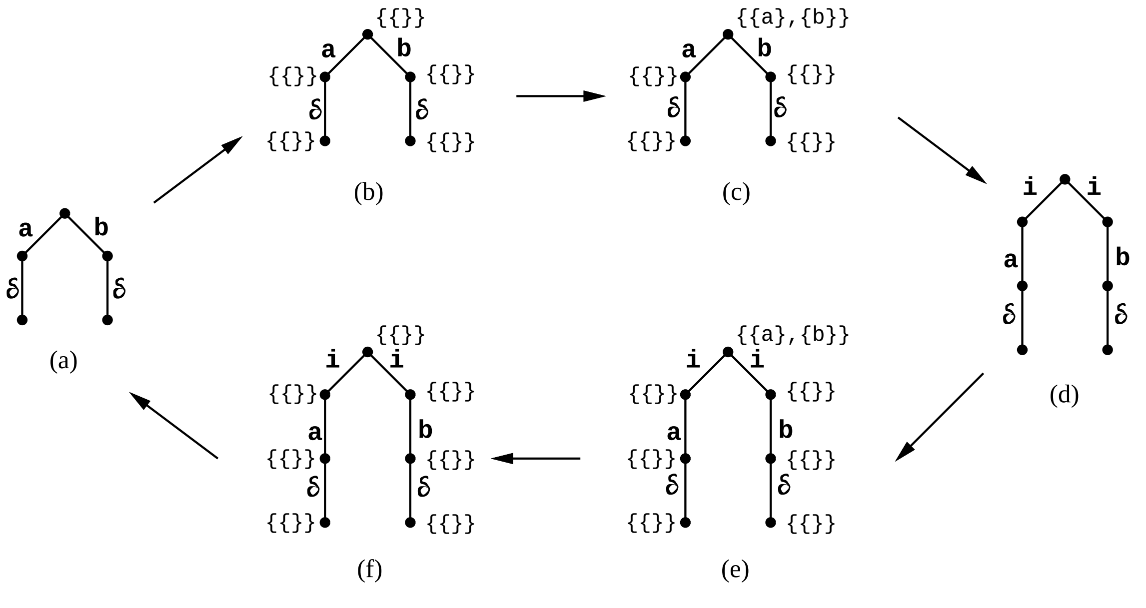


Figure 29: Generating canonical testers

4.3.6 Discussion

The TCG method for LOTOS discussed in this section is well-defined and has a very strong theoretical basis. However, it has the following drawbacks that make this method impractical.

1. It is based on basic LOTOS, which ignores the data aspect of the specification. This make the application of the method fairy restricted. Some approaches are suggested [18, 19, 22], but no solution is given.

2. The relationship **conf** is so strong that sometimes, a correct implementation may not be treated as a conforming implementation [3].

3. The test case generated by the method contains almost every possible trace of the specification. As shown in Section 4.1, the number of trances can be huge even in some small protocols. Fully test all traces is just not practical.

As the result, the method presented in this section can be viewed as a theoretical basis of conformance testing, but cannot serve as a practical TCG method.

4.4 Summary

As shown in Section 4.1, TCG for FSM is not sophisticated enough to handle real protocols. When the protocol specification gets larger, an FSM usually becomes too complicated for existing TCG methods to generate any practical test cases. Therefore, there is a need for a TCG method based on extended models that can handle the data aspect of a protocol specification.

A handful of proposals for TCG methods based on Estelle or LOTOS have been presented, recently. However, the TCG methods for Estelle shown above fail to establish usable test cases, and the methods for LOTOS are too theoretical to be used in practice. None of them actually generates step-by-step test cases that can be followed. Therefore, a novel approach that uses axiomatic semantics to deal with the problem in testing the data aspects will be proposed in the later chapters.

CHAPTER V

Axiomatic Test Case Generation (ATG)

As we reviewed previous work on TCG methods, none have yielded satisfactory results. The study on the FSM model is well established, but is difficult to use in practice. Work on extended models is still premature, and cannot generate actual test cases. Therefore, from Chapter V to Chapter IX, a sequence of TCG methods are proposed, which intend to generate test cases automatically for real-life protocols.

5.1 Overview of TCG for EFSM

In this and the following four chapters (Chapter V to IX), TCG methods for EFSM are described in detail. The first three methods (ATG, OAT, and TGF) are based on the EFSM model described in Section 2.3, and the forth is an extension of TGF using ISO standard specification language Estelle. The last one transforms TCG problem into a protocol validation problem and uses existing validation tools to generate test cases.

The ATG and the OAT methods (Chapters V and VI) are used to detect those implementation faults in which a transition is either (1) pointing to a wrong tail state, or (2) having an action that produces incorrect effects. The former is called a *transfer fault*, and the latter is called a *data fault*. A *fault model* specifies a class of errors that can happen in an EFSM. The fault model of an EFSM having only one erroneous transition that has either transfer fault or data fault, is called *single-transition faults.*

In the TCG methods proposed in this dissertation, the ATG and the OAT methods are used to detect single-transition faults; their detecting other fault models is possible but not guaranteed. The TGF method, on the other hand, is able to generate test cases for given fault models (Chapter VII). It is also extended to generate test cases for Estelle (Chapter VIII). Chapter IX uses the same principle of fault models, but solve it using protocol validation methods.

It is assumed that a specification has the following properties:

A1: There is a reset transition, through which an EFSM can enter the initial state from any state in the EFSM.

A2: Every variable used must contain a defined value. That is, no uncertain value is allowed.

A3: Once a transition is ready to fire, the execution time of any transition is bounded.

A4: The specification is *minimum*, meaning that the external behavior of a specification will change if any transition is removed.

A5: The state-transition diagram is *strongly connected*, meaning that there is a path between any two states.

A protocol without reset operation is hard to synchronize and difficult to maintain (assumption A1). Therefore, there is hardly any real-life protocol without any reset operations. If a protocol has a reset option, assumption A5 must be satisfied or the state-transition diagram would be partitioned, in which some states could not be reached from the initial state. Assumption A2 is feasible because a well behaved program should not use any undefined variable. It is also reasonable to assume that the given specification is minimum (assumption A4), for if it is not, one can always reduce it to a minimal specification and use the minimized specification as the input of the TCG methods. Finally, to avoid encountering some unsolvable problem (e.g., the Halting Problem), assumption A3 is used to set a limit on how far the TCG methods should continue before it can find an answer.

5.2 Basic Concept of Axiomatic Test Case Generation

Axiomatic semantics for program proving has been introduced in Section 2.5. The forward and backward axioms for assignment statements are designed differently to suit different approaches of proving processes. In fact, depending on how an axiom is designed, axiomatic semantics can serve other purposes rather than just used as a program proving tool. Figure 30 shows an example of how axiomatic semantics can be used for other purposes.

In Figure 30a, the axiom for the output statement appends the output to the precondition, while the axiom of the assignment statement keeps the postcondition the same as the precondition. The resulting effect is that the assertion now records a sequence of output events. For example, the verification of the program in Figure 30b is shown in Figure 30c. Initially, the assertion is an empty set (line 1). Applying the output axiom to the statement in line 2 results in the assertion in line 3, which contains

$\{p\}$ G!x; $\{p\ x\}$
$\{p\}$ x:=e; $\{p\}$

(a)

G!1;
x:=x+1;
G!2;

(b)

1. { }
2. G!1;
3. {1}
4. x:=x+1;
5. {1}
6. G!2;
7. {1 2}

(c)

Figure 30: An axiom that records the output sequence generated by a program. (a) Axioms for output and assignment statements, (b) A simple program, and (c) Evaluating the simple program.

the element just output. Then, the assignment axiom is applied to the statement in line 4 and obtains the same assertion in line 5. Finally, applying the output axiom again derives the final assertion in line 7, which contains all the outputs generated by this program.

The idea of axiomatic test case generation is that, perhaps, one can somehow design the axioms so that after the program is verified, the final assertion contains a test case. This idea is used in both this and the next chapters. In this chapter, a simple axiomatic TCG, called ATG, is described to generate test cases for each individual transition. In the next chapter (Chapter VI), a more complicated axiomatic TCG (called OAT) is introduced, which considers more than one transition at once. Since each test case generated by OAT detects errors for more than one transition, the overall length of the test suite will be much shorter than what ATG generates.

Researchers have proposed several program testing methods based on symbolic executions [33, 12]. However, their methods follow a totally different approach from those proposed in this dissertation. Their approaches are called *white box testing*, meaning that the source of the implementation is available. Their methods symbolically execute the implementations through all possible branches, statements, input and output data, values of variables, etc. Then, the user decides whether the implementation is correct based on the final assertions. However, the white box testing approach are not suitable for testing communication protocols because (1) their approaches do not take advantage of the specifications, which is usually clearly defined for communication protocols, (2) their approaches require human intuition to determine whether the final assertions conform with the expected, (3) the program source is not always available, and (4) a correct source program does not necessarily mean that the implementation will run correctly on any hardware or software environments (e.g., when the lower layer protocols do not provide necessary services). Moreover, their approaches mostly deal with computational type of program (e.g., scientific or financial calculation), few of them paid attention to the data processing (e.g., protocols). Therefore, the goal of this and the following chapters is to find a suitable TCG method for communication protocol, using axiomatic semantics as a tool.

5.3 Assertions and Axioms for ATG

To generate a test case, the assertions and axioms used must be more complicated than those described in Section 2.5. There are three fields in the assertions, namely: $\mathcal{E}$ (for Events), $\mathcal{P}$ (for Predicates), and $\mathcal{V}$ (for Variables) which are defined as follows.

1. Field $\mathcal{E}$ is a sequence of external events. This field records every external event appearing along the traversed path, so it can be used as a test case at the end of the evaluation. For example, an external event sequence can be "$\langle G?2, G!x \rangle$," meaning that input a "2" through gate G, and then output whatever value in x through the same gate.

2. Field $\mathcal{P}$ is a set of predicates, which determines the current status of an EFSM. For instance, a predicate can be "$\{x = y \wedge z > 1 \wedge \text{IsSorted}(A)\}$," indicating that at the current state, $x = y$, $z > 1$, and array A is sorted.

3. Field $\mathcal{V}$ is a set of variable groups that need to be *observed.* A variable is observed if its value is revealed by an output event appeared in field $\mathcal{E}$. Each variable group in $\mathcal{V}$ is quoted by a pair of angle brackets ($\langle\ \rangle$). For example, $\{\langle A, B \rangle, \langle C, D \rangle\}$ contains two variable groups, $\langle A, B \rangle$ and $\langle C, D \rangle$.

In a program, an assertion describes the conditions between two statements, but in an EFSM, an assertion represents the conditions at a state. An axiom is applied to the action of a transition, and the precondition and the postcondition are the assertions for the head state and the tail state of the transition, respectively.

Assuming that fields $\mathcal{E}$, $\mathcal{P}$, and $\mathcal{V}$ in a precondition of a transition are E, P, and V, respectively, an axiom should be defined so that the postcondition follows these principles:

1. If there are any variables being changed after the action, the original values of those variables in E, P and V become outdated. That is, E, P, and V should be changed into $E_{\nu^{-1}x_1,\nu^{-1}x_2,...,\nu^{-1}x_n}$, $P_{\nu^{-1}x_1,\nu^{-1}x_2,...,\nu^{-1}x_n}$, and $V_{\nu^{-1}x_1,\nu^{-1}x_2,...,\nu^{-1}x_n}$ respectively, assuming that $x_1, x_2, ..., x_n$ are the modified variables. (see Section 2.5 for the notation)

2. Any external events should be recorded in $\mathcal{E}$.

3. Any new condition that states the relationship among the variables should be included in $\mathcal{P}$.

4. Any variables whose values need to be observed should be grouped and inserted into $\mathcal{V}$.

Table 9 is an example of how the axioms of the actions in Figure 6 can be defined. There are four types of axioms, each of which represents a type of action. Assume that in the assertion before a transition, $\mathcal{E}$, $\mathcal{P}$, and $\mathcal{V}$, are equivalent to E, P, and V, respectively. After the transitions, $\mathcal{E}$, $\mathcal{P}$, and $\mathcal{V}$ will be changed according to the following in the postcondition.

Since ATG generates test case for each transition, the transition for which the test case is used is called the *transition under test (TUT)*. The axioms in Table 9 are explained in detail as follows.

Table 9: Axioms for ATG

Transitions	Axioms
Assignment $x := e$	$\mathcal{E}= E_{\nu^{-1}x}$ $\mathcal{P}= P_{\nu^{-1}x} \cup \{x = e_{\nu^{-1}x}\}$ $\mathcal{V}= \begin{cases} V_{\nu^{-1}x} \cup \{\langle x \rangle\} & \text{if TUT} \\ V_{\nu^{-1}x} & \text{otherwise} \end{cases}$
Condition $\mathcal{B}$	$\mathcal{E}= E$ $\mathcal{P}= P \cup \{\mathcal{B}\}$ $\mathcal{V}= V$
Input $G?x$	$\mathcal{E}= E_{\nu^{-1}x} + \langle G?x \rangle$ $\mathcal{P}= P_{\nu^{-1}x} \cup \{x = m\}$ $\mathcal{V}= \begin{cases} V_{\nu^{-1}x} \cup \{\langle x \rangle\} & \text{if TUT} \\ V_{\nu^{-1}x} & \text{otherwise} \end{cases}$
Output $G!e$	$\mathcal{E}= E + \langle G?e \rangle$ $\mathcal{P}= P$ $\mathcal{V}= V$

Assignment Axiom: Suppose that an assignment action assigns the result of expression e to variable x. Since the values of x before and after the transition are different, every occurrence of x in the original assertion is replaced by $\nu^{-1}x$ (denoted by $E_{\nu^{-1}x}$, $P_{\nu^{-1}x}$, and $V_{\nu^{-1}x}$).

Since assignment is an internal action, no new event is added to $\mathcal{E}$. After the action, variable x will be equal to the result of expression e, except that if e retrieves a value from x, it retrieves an outdated version. Therefore, "$x = e_{\nu^{-1}x}$" is inserted into $\mathcal{P}$. If the transition associated with the action is a TUT, the observation of variable x can determine whether the result of expression e is correctly assigned to x. Thus, $\langle x \rangle$ is put into $\mathcal{V}$, so that ATG can check for any clues in the external events to determine whether x has the expected value.

Condition Axiom: For a conditional action to be executed, condition $\mathcal{C}$ of the transition must be satisfied. Therefore, $\mathcal{C}$ is a new information obtained by the execution of the transition, and should be inserted into $\mathcal{P}$.

Input Axiom: An input transition is similar to an assignment transition; both of them assign a value to variable x. The difference is that an assignment transition computes the value from an expression, while the input transition receives the value from a gate. Therefore, the input axiom is very similar to the assignment axiom except that an input transition is external and should be inserted into $\mathcal{E}$. The value κ in the assertion is an arbitrary constant received from gate G.

Output Axiom: An output transition is an observable event. Therefore, it should be inserted into $\mathcal{E}$. Everything else remains unchanged.

Figure 31 is an example that applies the input axiom to transition 4 of the EFSM in Figure 6. Suppose that before the transition is executed, the assertion is the one shown at the left-hand-side of the arrow. According to the input axiom, every occurrence of A in the precondition should be replaced by $\nu^{-1}A$. In addition, $R?A$ and $A = \kappa_2$ are inserted into $\mathcal{E}$ and $\mathcal{P}$, respectively, where κ_2 is an arbitrary input constant. Assume that the transition is a TUT. According to the axiom, variable group $\langle A \rangle$ should be included in $\mathcal{V}$ so that whenever it is observed, ATG knows that the input value κ_1 is correctly assigned. The resulting assertion is shown at the right-hand-side of the arrow.

$\mathcal{E}= \langle R?A, R!(M,B)\rangle$ $\mathcal{P}= \{A = \kappa_1 \wedge A \neq B\}$ $\mathcal{V}= \{\langle A\rangle\}$ 1	$\xrightarrow{4\,:\,R?A}$	$\mathcal{E}= \langle R?\nu^{-1}A, R!(M,B), R?A\rangle$ $\mathcal{P}= \{\nu^{-1}A = \kappa_1 \wedge \nu^{-1}A \neq B \wedge A = \kappa_2\}$ $\mathcal{V}= \{\langle \nu^{-1}A\rangle, \langle A\rangle\}$ 2

Figure 31: ATG input axiom applies to transition 4 in Figure 6

5.4 Test Case Generation

A transition is considered correct if it performs a correct action and goes to an expected state. Therefore, to test a transition, one must ensure that the tail state is reached and the action is correctly performed. Procedures that test the correctness of the tail state and the associated action of a TUT are proposed in Section 5.4.2 and Section 5.4.3 respectively. Both procedures use the search procedure described in Section 5.4.1 to find paths that satisfy some necessary properties. Then, the procedures are combined into a test case generation algorithm described in Section 5.4.4.

5.4.1 Searching for a Path

The algorithms provided in this paper frequently need to search for a specific path that satisfies a certain property. This can be achieved by performing Procedure **Forward-Search** in Figure 32. Given a path P and an assertion A, the algorithm extends the path until the given property is satisfied. It is, in fact, a breadth-first search among all possible paths in an EFSM.

Procedure Forward-Search

description:

This procedure search forward through the state diagram of an EFSM to find a path that satisfies a certain property. The satisfactory of the property is determined by a function passed in as a parameter. The function takes a path and an assertion from this procedure, and the rest of the argument form the caller of this procedure. The function returns a **true** if the property is met.

input:

M: an EFSM
P: a path,
A: an assertion, and
F: a procedure determining the satisfactory of the given property

output:

an extended path and
the assertion of the path's tail state

Put pair (P, A) in a queue Q
while Q is not empty **do**
    Get a pair (P', A') from Q
    Let s be the tail state of P'
    for every outgoing transition t of s **do**
        Apply appropriate axiom of t to A'
        Let A' the resulting assertion, and
        Append t the end of P'
        if $F(P', A', ...)$ returns true **then**
            return P' and A'
        end-if
    (1) **if** P' is not too long **then**
            Put (P', A') into Q
        end-if
    end-for
end-while
return path not found

Figure 32: Procedure **Forward-Search**

In general, deciding whether a Turing-equivalent model does or does not have a certain property is an unsolvable problem [39]. That is, one can never tell whether a path with a desired property can ever be found. Therefore, a limit must be set so a path will not be extended forever. This is handled by the step labeled *(1)* in Procedure **Forward-Search**. Even if the path does exist, there are no efficient algorithms available to find it. The only way to find a path with a given property is through exhaustive search. However, heuristics can be used to enhance the efficiency of the search and some are suggested in Chapter VIII.

5.4.2 Confirming the Tail State

Two states are said to be *indistinguishable* from each other, if (1) they are the head states of some external transitions that input or output through the same gate, or (2) one state is indistinguishable from the tail state of any outgoing internal transitions of the other state. In other words, two states are indistinguishable if no one can tell which is which from the next external event. For example, in Figure 6, state a and state f are indistinguishable because their next external events ($U?M$) are both input actions at gate U.

A *mutant implementation*, or simply a *mutant*, is an EFSM that differs from the original EFSM by a single transition, called a *mutant transition.* Instead of pointing to a correct state, a mutant transition points to another state that is indistinguishable from the correct one. For instance, Figure 33a is a mutant of the EFSM in Figure 6. The mutant transition, transition 4, points to state c, which is indistinguishable from state e (the correct state).

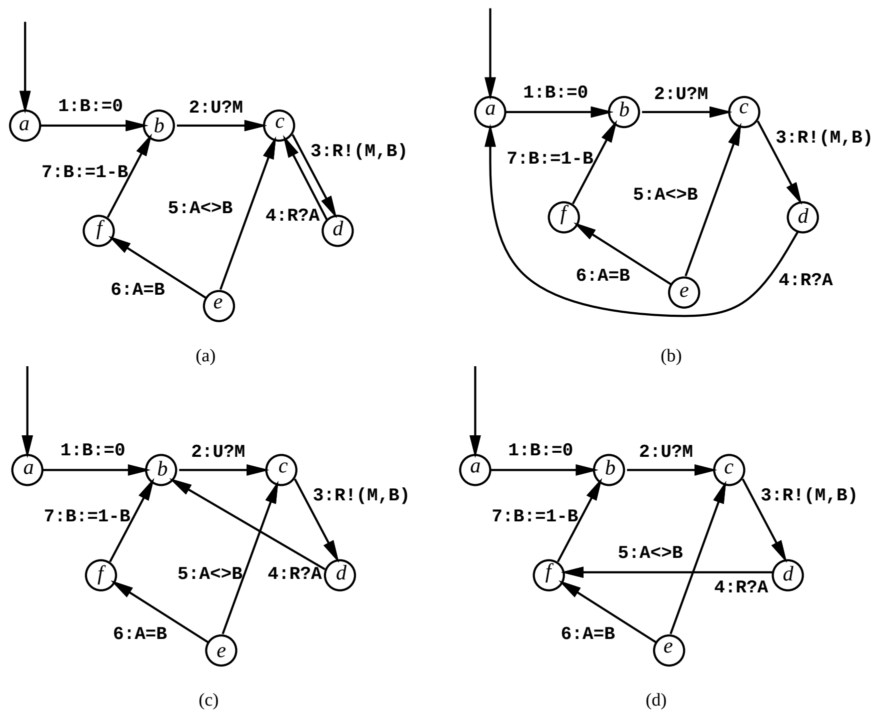


Figure 33: Four mutants with erroneous transition 4

Two states can be *differentiated* if there exists an external event sequence that reveals the EFSM originally being in one state but not in the other. (This is similar to the UIO sequence that distinguishes one state from every other state in an FSM.) It is easy to differentiate those states that are distinguishable; the next external event will reveal the difference of the states (or they will be indistinguishable in the first place). For those indistinguishable states, it is necessary to find an external event sequence that can be produced by one state but not the other. (A state can "produce" an external event sequence if the external event sequence is generated by a legal path headed by that state.) Therefore, to tell whether a TUT actually reaches a correct tail state, one must find a path whose external event sequence cannot be reproduced by the mutants. The path that satisfies the criteria is called a *state confirming path*, which can be found by the following procedures. Procedure **Confirm-State** generates a sequence of external events for the specification, and impose the sequence to the mutants by calling **Impose-To**. The procedure generates one external event at a time, and remove those mutants that cannot generate the event sequence. When all the mutants are removed, the resulting sequence can be used as a state confirming path.

5.4.3 Confirming the Action

An action can be confirmed if the effects caused by the action can be observed from an external event sequence. If a transition changes the contents of some variables, these variables must be observed to ensure that the changes are correct. Therefore, those variables being changed are grouped and included in field $\mathcal{V}$. A variable can

Procedure Find-External

description:
This procedure determines whether P_2 generates one more external events than P_1.

input:
P_1: a path
A_1: an assertion of P_1's tail state
P_2: another path
A_2: an assertion of P_2's tail state

output:
true or false

if field $\mathcal{E}$ in A_1 contains one more element then field $\mathcal{E}$ in A_2 **then**
  return true
else
  return false
end-if

Figure 34: Procedure **Find-External**

Procedure Impose-To

description:
This procedure check if a given event sequence can be generated by a given EFSM.

input:
E: an external event sequence
M: an EFSM
t: the transition in M where the checking should start

output:
true or false

```
Apply the axiom for t to an empty assertion
Let the resulting assertion be A
for every next event e in E do
    Call Forward-Search(⟨t⟩,A,Find-External(⟨t⟩, A))
    if no path found then
        return false
    else
        Let A' be the resulting assertion obtained by Forward-Search
        Let e' be the last external event in the ℰfield in A'
        if e and e' are not the same then
            return false
        end-if
    end-if
end-for
return true
```

Figure 35: Procedure **Impose-To**

Procedure Confirm-State

description:
This procedure generates a state confirming path

input:
P: a path A: an assertion

output:
A state confirming path, and
An assertion associated with the tail state of confirming path

while P is not too long **do**
  Call **Forward-Search**(P, A, **Find-External**(P, A))
  Let $\mathcal{M}$ be a set of mutants
  Let P be the resulting path
  Let A be the resulting assertion
  Let E be the $\mathcal{E}$ field in A
  for every mutant M in $\mathcal{M}$ **do**
    Let t_m be the mutant transition of M
    if Impose-To(E, M, t_m) returns false **then**
      Remove M from $\mathcal{M}$
    end-if
  end-for
  if $\mathcal{M} = \emptyset$ **then**
    return P and A
  end-if
end-while
return path not found

Figure 36: Procedure **Confirm-State**

be *observed* if either (1) it appears in some output events, or (2) another variable whose value depends on the former variable appearing in some output events. That is, a variable can be observed if either (1) it can be observed directly, or (2) its value is propagated to other variables that can be observed. The information of which variables depend on the variable needed to observed can be found from the predicate in field $\mathcal{P}$. If every variable in any variable group of field $\mathcal{V}$ can be observed, the field is said to be *observable*. For example, if $\mathcal{V}= \{\langle x, y\rangle, \langle z, w\rangle\}$, $\mathcal{V}$ is observable if either both x and y, or both z and w can be observed. A path is called an *action confirming path* if its external event sequence confirms the correctness of the modified variables. Procedure **Confirming-Action** in Figure 38 describes how to search for such a path.

Procedure Observable

description:
Check if assertion A is observable

input:
P: a path
A: an assertion of P_1's tail state

output:
true or false

if $\mathcal{V}$ in A is observable **then**
  return true
else
  return false
end-if

Figure 37: Procedure **Observable**

Procedure Confirming-Action

description:
Search for an action confirming path

input:
P: a path
A: an assertion

output:
An action confirming path, and
The assertion of the path's tail state

if Observable($-$, A) returns true **then**
    return P and A
end-if
Call **Forward-Search**(P, A, **Observable**())
Let the resulting path and assertion be P and A, respectively
return P and A

Figure 38: Procedure **Confirming-Action**

5.4.4 Generating a Test Case

Recall that a transition is considered correct if both its tail state and its action are confirmed. Therefore, a test case of a TUT must include both the state confirming path and the action confirming path of the TUT. This can be done by the following steps. First, generate the state confirming path for the TUT. Then, extend the path until an action confirming path is found. The resulting path is called a *test path* and field $\mathcal{E}$ in the final assertion is the test case. The value of the variable in $\mathcal{E}$ can be found in field $\mathcal{P}$. The procedure generating a test case is shown in Figure 39.

Procedure Test-Case

description:
This procedure generates a test case for a transition.

input:
t: a transition

output:
A test path, and
An assertion of the tail state of the path

Apply a proper axiom for t to an empty assertion
Let the resulting assertion be A
Call **Confirm-State**($\langle t \rangle$, A)
Let the resulting path be P and the assertion be A
Call **Confirming-Action**(P, A)
return the resulting path and the resulting assertion

Figure 39: Procedure **Test-Case**

5.5 Examples

Figure 40 shows an example of how the test case for transition 2 in Figure 6 can be constructed. In Figure 40a, the state confirming path is immediately obtained since action $U?M$ is the only transition that inputs a value from gate U. The precondition is shown at the left-hand-side of the arrow (in which all fields are empty), and the postcondition is shown at the right. In Figure 40b, starting from the tail state and the postcondition of the state confirming path, an action confirming path can be found through path $\langle 3 \rangle$. Field $\mathcal{V}$ of the resulting assertion contains a variable group, $\langle M \rangle$. Since variable M appears in an output event, "$R!(M, B)$," of field $\mathcal{E}$, whether the value in M is correct can be observed directly from the outcome of the event.

Therefore, the correctness of action $U?M$ can be confirmed. As a result, the test path for transition 2 is $\langle 2, 3\rangle$ and the test case is $\langle U?m, R!(m, B)\rangle$ (because of "$M = m$" in $\mathcal{P}$). The notation of the test case means that first, the IUT expects an input from gate U, and then it sends an output to gate R. The message part of the output should be the same as the previous input, and the message sequence number is whatever value contained in variable B, which is unimportant to this test case.

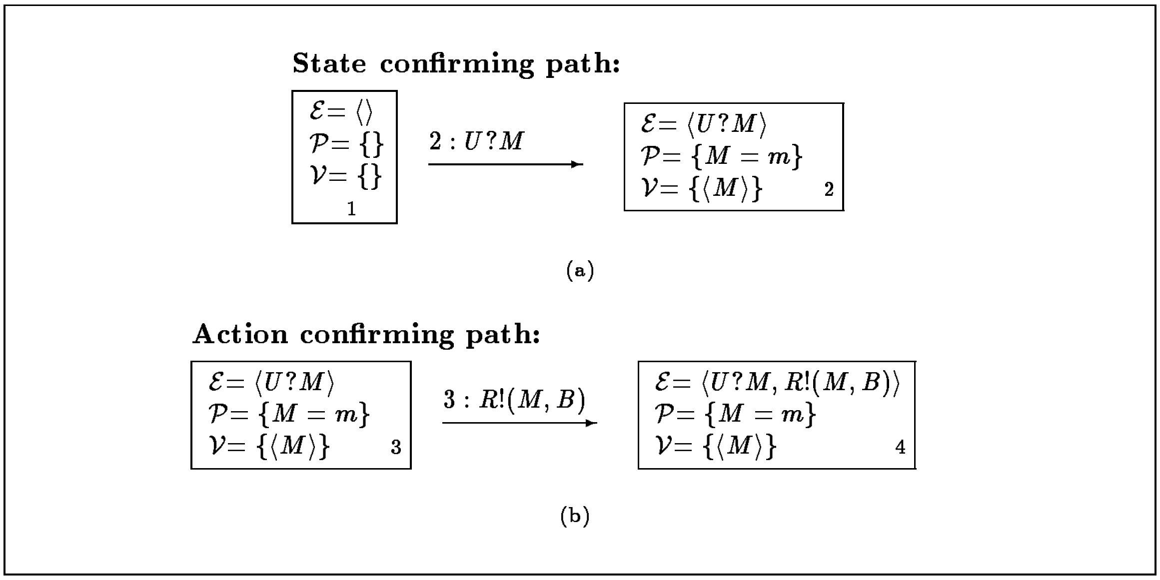


Figure 40: Test paths for transition 2 of Figure 6. (a) Generating state confirming path, and (b) generating action confirming path

Figure 41 is a more complicated example, which shows how the test case of transition 4 is derived. In the figure, the tail state of transition 4 (state e) cannot be distinguished from states a, b, c, and f. Therefore, the mutants are constructed and shown in Figure 33. A state confirming path that differentiates these mutants is $\langle 4, 5, 3, 4, 6, 7\rangle$, which is derived in Figure 41. Field $\mathcal{V}$ of the last assertion is observable because (1) "$\langle A \rangle$" in $\mathcal{V}$, meaning that if A is observed, the action of transition 4 can be confirmed, (2) "$A = \nu^{-1}B$" in $\mathcal{P}$, meaning that A and $\nu^{-1}B$ are equivalent, and (3) "$R!(\nu^{-1}M, \nu^{-1}B)$" in $\mathcal{E}$, meaning that $\nu^{-1}B$ can be observed, so A can be observed as well. That is, the value of A can be observed indirectly from the observation of $\nu^{-1}B$. Therefore, no action confirming path is needed. The test case for transition 4 is $\langle R?\kappa_1, R!(\nu^{-1}M, \kappa_2), R?\kappa_2, U?\kappa_3 \rangle$, where $\kappa_1 \neq \kappa_2$ (since $\nu^{-1}A = \kappa_1$, $A = \kappa_2$, $A = \nu^{-1}B$, and $\nu^{-1}A \neq \nu^{-1}B$). Note that $\nu^{-1}M$ does not equal to any constants, which means that its value is irrelevant to this test case and can be arbitrary.

5.6 Combine Test Cases into a Test Suite

To combine all the test cases to form a test suite, some "glue" is needed between two test cases. However, some test paths are legal only if certain conditions hold. For example, the postcondition of the test path for transition 5 (shown as the last assertion in Figure 42) shows that to start the test cases, variable A must not equal to B. Otherwise, the test case can never be executed. Therefore, to connect the test case, one must be careful not to violate the necessary condition for a test case. A complete test suite can be constructed by the procedure in Figures 43 and 44.

Figure 45 shows the complete test sequence for the Alternating Bit Protocol, where the sequences quoted by square brackets ([ ]) are test cases. The I/O events between two test cases are used to transit the EFSM from one test case to another. Variables κ_1 to κ_9 are the input values, and the r in the sequence denotes the reset signal.

State confirming path:

$\mathcal{E}=\langle\rangle$
$\mathcal{P}=\{\}$
$\mathcal{V}=\{\}$
1

$\xrightarrow{4\,:\,R?A}$

$\mathcal{E}=\langle R?A\rangle$
$\mathcal{P}=\{A=\kappa_1\}$
$\mathcal{V}=\{\langle A\rangle\}$
2

$\xrightarrow{5\,:\,A\neq B}$

$\mathcal{E}=\langle R?A\rangle$
$\mathcal{P}=\{A=\kappa_1\wedge A\neq B\}$
$\mathcal{V}=\{\langle A\rangle\}$
3

$\xrightarrow{3\,:\,R!(M,B)}$

$\mathcal{E}=\langle R?A, R!(M,B)\rangle$
$\mathcal{P}=\{A=\kappa_1\wedge A\neq B\}$
$\mathcal{V}=\{\langle A\rangle\}$
4

$\xrightarrow{4\,:\,R?A}$

$\mathcal{E}=\langle R?\nu^{-1}A, R!(M,B), R?A\rangle$
$\mathcal{P}=\{\nu^{-1}A=\kappa_1\wedge\nu^{-1}A\neq B$
$\wedge A=\kappa_2\}$
$\mathcal{V}=\{\langle\nu^{-1}A\rangle,\langle A\rangle\}$
5

$\xrightarrow{6\,:\,A=B}$

$\mathcal{E}=\langle R?\nu^{-1}A, R!(M,B), R?A\rangle$
$\mathcal{P}=\{\nu^{-1}A=\kappa_1\wedge\nu^{-1}A\neq B\wedge$
$A=\kappa_2\wedge A=B\}$
$\mathcal{V}=\{\langle\nu^{-1}A\rangle,\langle A\rangle\}$
6

$\xrightarrow{7\,:\,B:=1-B}$

$\mathcal{E}=\langle R?\nu^{-1}A, R!(M,\nu^{-1}B), R?A\rangle$
$\mathcal{P}=\{\nu^{-1}A=\kappa_1\wedge\nu^{-1}A\neq\nu^{-1}B\wedge$
$A=\kappa_2\wedge A=\nu^{-1}B\wedge B=1-\nu^{-1}B\}$
$\mathcal{V}=\{\langle\nu^{-1}A\rangle,\langle A\rangle\}$
7

$\xrightarrow{2\,:\,U?M}$

$\mathcal{E}=\langle R?\nu^{-1}A, R!(\nu^{-1}M,\nu^{-1}B), R?A, U?M\rangle$
$\mathcal{P}=\{\nu^{-1}A=\kappa_1\wedge\nu^{-1}A\neq\nu^{-1}B\wedge A=\kappa_2\wedge$
$A=\nu^{-1}B\wedge B=1-\nu^{-1}B\wedge M=\kappa_3\}$
$\mathcal{V}=\{\langle\nu^{-1}A\rangle,\langle A\rangle\}$
8

Figure 41: Test path for transition 4 in Figure 6

State confirming path:

$\mathcal{E}=\langle\rangle$
$\mathcal{P}=\{\}$
$\mathcal{V}=\{\}$
1

$\xrightarrow{5\,:\,A\neq B}$

$\mathcal{E}=\langle\rangle$
$\mathcal{P}=\{A\neq B\}$
$\mathcal{V}=\{\}$
2

$\xrightarrow{3\,:\,R!(M,B)}$

$\mathcal{E}=\langle R!(M,B)\rangle$
$\mathcal{P}=\{A\neq B\}$
$\mathcal{V}=\{\}$
3

Figure 42: Test path for transition 5 in Figure 6

Procedure Link

description:
This procedure checks if a given path reaches the head state of another path

input:
P_1: a path given by **Forward-Search**
A_1: an assertion given by **Forward-Search**
P_2: a path
A_2: an assertion

output:
true or false

if $P_1 = P_2$ and fields $\mathcal{P}$'s in A_1 and A_2 does not conflict **then**
    return true
else
    return false
end-if

Figure 43: Procedure **Link**

5.7 Summary

ATG, the first TCG method proposed in this dissertation, is proposed in this chapter. The method provides a set of assertion to store necessary information for generating test cases. Axioms are designed for each type of statement to transform one assertion to another. ATG generates a test case for each transition by generating state confirming and action confirming paths. A state confirming path confirms the correctness of the transition's tail state, while the action confirming path detects the incorrect effect of the transition's action. The combination of the state and action confirming

Procedure Test-Suite

description:
This procedure generates test suite for an EFSM

input:
M: an EFSM

output:
A test suite

Let s be the initial state of M
Let A be an empty assertion
Let T be an empty sequence
for every transition t in M **do**
Call **Test-Case**(t) to get a test path P and an assertion A
Call **Forward-Search**($\langle r \rangle$, A, **Link**(P,A))
Let the resulting path be P'
Append the P' to T, and then append P to T
end-for **return** T

Figure 44: Procedure **Test-Suite**

$\langle \quad r, [U?\kappa_1, R!(\kappa_1, 0)],$	(for transition 1)
$r, [U?\kappa_2, R!(\kappa_2, 0)],$	(for transition 2)
$r, U?\kappa_3, [R!(\kappa_3, 0)],$	(for transition 3)
$r, U?\kappa_4, R!(\kappa_4, 0), [R?1, R!(\kappa_4, 0), R?0, U?\kappa_5],$	(for transition 4)
$R!(\kappa_5, 1), R?0, [R!(\kappa_5, 1)],$	(for transition 5)
$R?1, [U?\kappa_6, R!(\kappa_6, 0), R?0, U?\kappa_7, R!(\kappa_7, 1)],$	(for transition 6)
$R?1, U?\kappa_8, [R!(\kappa_8, 0), R?0, U?\kappa_9, R!(\kappa_9, 1)] \rangle$	(for transition 7)

Figure 45: A complete test sequence for Alternating Bit Protocol

paths makes a test path, whose external events can be used as a test case. The test cases for all the transitions in an EFSM are connected to form a test suite.

Since each test case generated by ATG is derived for each individual transition, a test case is very likely to be redundant or to overlap other test cases. It requires a more sophisticated method to eliminate the redundancies. Therefore, in the next chapter, a more complicated method, call OAT, is introduced to solve this problem.

CHAPTER VI

Optimized Axiomatic Test Case Generation (OAT)

The ATG method proposed in the previous chapter generates test cases for each transition individually, and the test suite is composed by concatenating every test case together. Since generating a test case for a transition does not take into consideration other transitions' test cases, one can expect that there will be much redundancy among the test cases. For example, in Figure 45, the test cases for transitions 1, 2, and 3 are exactly the same; all reset the EFSM, get an input from gate U, and then output the input value to gate R with sequence number 0. Moreover, the first three events of the test case for transition 4 are also identical to the test cases for transition 1, 2, and 3, and the last three events of the test case overlaps with the test case of transition 5. Such redundancy and overlapping should be eliminated so the size of the final test suite can be shortened.

This chapter discusses another axiomatic TCG method, called *OAT*, which considers more than one transition at once in order to minimize the length of the overall test suite. As the result, the assertions and axioms used in OAT are more complicated then those in the ATG method. Moreover, the OAT method utilizes both backward and forward evaluations, which makes the algorithm even more complicated. However, OAT does generate much shorter test suite than ATG does, which pays off its complexity.

6.1 Assertions and Axioms for OAT

To simultaneously generate a test case that detects errors for more than one transition, more information in the assertion is needed. In fact, there are eight fields in an assertion, which are described as follows.

1. Field $\mathcal{E}$ is a sequence of external events.
2. Field $\mathcal{P}$ contains the predicate that indicates the current status of an EFSM. The purpose of the first two fields is similar to the $\mathcal{E}$ and $\mathcal{P}$ fields used in the ATG method (page 95).
3. Field $\mathcal{D}$ contains a set of variables that have been *defined.* A variable is defined if its value has been known.
4. Field $\overline{\mathcal{D}}$, on the other hand, contains a set of variables that have not been defined.
5. Field $\mathcal{A}$ contains a set of transitions with *confirmed* actions. An action is confirmed if the external events of the path traversed by OAT reveals the action's correctness.
6. Field $\overline{\mathcal{A}}$ contains a set of transitions with unconfirmed actions, each of which is attached with the variables modified by the action.

7. Field $\mathcal{S}$ contains a set of transitions with *confirmed* tail state. A tail state of a transition is confirmed if the path traversed by OAT reveals that the transition indeed reaches the state.

8. Field $\overline{\mathcal{S}}$ contains a set of transitions with unconfirmed tail states.

Similar to the ATG method, field $\mathcal{E}$ records the external events that will be used as a test case, and field $\mathcal{P}$ monitors the values of the variables. Fields $\mathcal{D}$ and $\overline{\mathcal{D}}$ are used in a backward evaluation to find a path that assigns each variables in the path with a value. Fields $\mathcal{A}$, $\overline{\mathcal{A}}$, $\mathcal{S}$, and $\overline{\mathcal{S}}$, on the other hand, are used in a forward evaluation. The former two are used to search for an action confirming path and the latter two are used to find a state confirming path.

As mentioned above, the OAT method uses both forward and backward evaluations to find a test case. The axioms are shown in Table 10. It is assumed that before the axioms are applied, fields $\mathcal{E}$, $\mathcal{P}$, $\overline{\mathcal{D}}$, $\mathcal{D}$, $\overline{\mathcal{A}}$, $\mathcal{A}$, $\overline{\mathcal{S}}$, and $\mathcal{S}$ in the original assertion (i.e., the post-condition for the backward axioms or the precondition for the forward axioms) are E, P, $\overline{D}$, D, $\overline{A}$, A, $\overline{S}$, and S, respectively. The meaning of the axioms in the table is described as follows. The assignment axioms are explained in detail, and the rest of the axioms follow a similar principle.

Backward Assignment Axiom: Since the value of variable x is changed after executing transition T, every occurrence of x in E, P, $\overline{D}$, D, and $\overline{A}$ is upgraded to νx (denoted $E_{\nu x}$, $P_{\nu x}$, $\overline{D}_{\nu x}$, $D_{\nu x}$, and $\overline{A}_{\nu x}$, respectively). (Fields $\mathcal{A}$, $\overline{\mathcal{S}}$, and $\mathcal{S}$ do not contain any variable; therefore, no version upgrading is needed.) That

Table 10: Axioms for OAT method

	Backward Axioms	Forward Axioms
Assignment $T : x := e$	$\mathcal{E}= E_{\nu x}$ $\mathcal{P}= P_{\nu x} \cup \{\nu x = e\}$ $\overline{\mathcal{D}}= \overline{D}_{\nu x} \cup \text{var}(e) - \{\nu x\}$ $\mathcal{D}= D_{\nu x} \cup \{\nu x\}$ $\overline{\mathcal{A}}= \overline{A}_{\nu x} \cup F_1$ $\mathcal{A}= A$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$	$\mathcal{E}= E_{\nu^{-1}x}$ $\mathcal{P}= P_{\nu^{-1}x} \cup \{x = e_{\nu^{-1}x}\}$ $\overline{\mathcal{D}}= \overline{D}_{\nu^{-1}x} \cup \text{var}(e_{\nu^{-1}x}) - D_{\nu^{-1}x}$ $\mathcal{D}= D_{\nu^{-1}x} \cup \{x\}$ $\overline{\mathcal{A}}= \overline{A}_{\nu^{-1}x} \cup F_2$ $\mathcal{A}= A$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$
Condition $T : \mathcal{C}$	$\mathcal{E}= E$ $\mathcal{P}= P \cup \{\mathcal{C}\}$ $\overline{\mathcal{D}}= \overline{D} \cup \text{var}(\mathcal{C})$ $\mathcal{D}= D$ $\overline{\mathcal{A}}= \overline{A}$ $\mathcal{A}= A \cup \{T\}$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$	$\mathcal{E}= E$ $\mathcal{P}= P \cup \{\mathcal{C}\}$ $\overline{\mathcal{D}}= \overline{D} \cup \text{var}(\mathcal{C}) - D$ $\mathcal{D}= D$ $\overline{\mathcal{A}}= \overline{A}$ $\mathcal{A}= A \cup \{T\}$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$
Input $T : G?x$	$\mathcal{E}= \langle G?\nu x\rangle + E_{\nu x}$ $\mathcal{P}= P_{\nu x} \cup \{\nu x = \kappa\}$ $\overline{\mathcal{D}}= \overline{D}_{\nu x} - \{\nu x\}$ $\mathcal{D}= D_{\nu x} \cup \{\nu x\}$ $\overline{\mathcal{A}}= \overline{A}_{\nu^{\cup}x} F_1$ $\mathcal{A}= A$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$	$\mathcal{E}= E_{\nu^{-1}x} + \langle G?x\rangle$ $\mathcal{P}= P_{\nu^{-1}x} \cup \{x = \kappa\}$ $\overline{\mathcal{D}}= \overline{D}_{\nu^{-1}x}$ $\mathcal{D}= D_{\nu^{-1}x} \cup \{x\}$ $\overline{\mathcal{A}}= \overline{A}_{\nu^{-1}x} \cup F_2$ $\mathcal{A}= A$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$
Output $T : G!e$	$\mathcal{E}= \langle G!e\rangle + E$ $\mathcal{P}= P$ $\overline{\mathcal{D}}= \overline{D} \cup \text{var}(e)$ $\mathcal{D}= D$ $\overline{\mathcal{A}}= \overline{A}$ $\mathcal{A}= A \cup \{T\}$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$	$\mathcal{E}= E + \langle O!e\rangle$ $\mathcal{P}= P$ $\overline{\mathcal{D}}= \overline{D} \cup \text{var}(e) - D$ $\mathcal{D}= D$ $\overline{\mathcal{A}}= \overline{A}$ $\mathcal{A}= A \cup \{T\}$ $\overline{\mathcal{S}}= \overline{S} \cup \{T\} - S$ $\mathcal{S}= S$

$$F_1 = \begin{cases} \{T(\nu x)\} & \text{if } (T \notin A) \\ \{\} & \text{otherwise} \end{cases} \qquad F_2 = \begin{cases} \{T(x)\} & \text{if } T \notin A \\ \{\} & \text{otherwise} \end{cases}$$

is, the original name x is used to represent an older version of variable x in transition T's precondition, and a new version νx is used to represent the value of x in the post-condition. The changes of the fields for this axiom are described as follows.

$\mathcal{E}$: Since assignment is an internal action, except upgrading variable x to νx, this field keeps unchanged.

$\mathcal{P}$: In addition to upgrading variable x in P, the predicate, $\nu x = e$ (meaning that the new version of x equals to the result of e), is a condition obtained by the execution of transition T, so it is included in $\mathcal{P}$.

$\overline{\mathcal{D}}$ and $\mathcal{D}$: Fields $\overline{\mathcal{D}}$ and $\mathcal{D}$ are used to store the information about those variables that need to be defined and whether they have or have not been defined. In the assignment statement, the result of expression e is assigned to variable x. Therefore, νx contains a known value and should be placed in $\mathcal{D}$ (denoted $D_{\nu x} \cup \{\nu x\}$). Meanwhile, since νx has been defined, if it exists in $\overline{D}_{\nu x}$, it should be removed.

If expression e contains any variables, those variables need to be defined or the value of x could not be known. The function var( ) in the table takes an expression and returns a set of variables that are used in the expression. Therefore, var(e) is included in $\overline{\mathcal{D}}$.

$\overline{\mathcal{A}}$ and $\mathcal{A}$: Field $\overline{\mathcal{A}}$ contains those transitions whose actions' correctness has not yet been confirmed. Along with each transition in $\overline{\mathcal{A}}$ is a set of variable whose values have been modified by the transition's action. Once the action of a transition has been confirmed, the transition is moved to $\mathcal{A}$.

When the assignment statement, $x = e$, is executed, the new value of x needs to be confirmed. Hence, $T(\nu x)$ is added to $\overline{\mathcal{A}}$, indicating that the action of transition T needs to be confirmed and the observation of νx, shall it happen, will reveal the correctness of the action. An exception is that, when the action of the transition has already been confirmed ($T \in A$), there is no need to confirm the same action again. If this is the case, transition T will not be added.

$\overline{\mathcal{S}}$ and $\mathcal{S}$: Fields $\mathcal{S}$ and $\overline{\mathcal{S}}$, respectively, are used to record which transition has a confirmed tail state and which does not. Therefore, traversed transition whose tail state has not yet been confirmed (not in S) will be added to $\overline{\mathcal{S}}$(this is achieved by adding the transition to set $\overline{S}$ and then remove those that are in S).

Forward Assignment Axiom: Similar to the backward assignment axiom, because the values of x before and after the transition T are different, any occurrence of x in the precondition is replaced by $\nu^{-1}x$. That is, the original name, x, is used to indicate the value of x before the transition, and a new version $\nu^{-1}x$ is used for the value of x in the post-condition. Therefore, E, P, $\overline{D}$, D, and $\overline{A}$, are changed to $E_{\nu^{-1}x}$, $P_{\nu^{-1}x}$, $\overline{D}_{\nu^{-1}x}$, $D_{\nu^{-1}x}$, and $D_{\nu^{-1}x}$, respectively. The changes of the fields in the assertion are described as follows.

$\mathcal{E}$: Similar to the backward axiom, the assignment action is an internal action, so that except for downgrading the original x in E to $\nu^{-1}x$, $\mathcal{E}$ field is not changed.

$\mathcal{P}$: After the assignment action is executed, the new value of x is equal to the result of expression e. However, if there is any reference to variable x in the expression, the value of x comes from the previous version. Therefore, predicate $x = e_{\nu^{-1}x}$ is added to $\mathcal{P}$.

$\overline{\mathcal{D}}$ and $\mathcal{D}$: Similar to the backward axiom, if there are any variables used by expression e, the variables need to be defined and is included in $\overline{\mathcal{D}}$(denoted $\overline{D}_{\nu^{-1}x} \cup \text{var}(e_{\nu^{-1}x})$). However, for those variables that have already been defined, there is no need to put them into $\overline{\mathcal{D}}$. Therefore, those variables that are in $\mathcal{D}$ are removed from $\overline{\mathcal{D}}$. On the contrary, since x is defined by the assignment action, it is included in $\mathcal{D}$.

$\overline{\mathcal{A}}$ and $\mathcal{A}$: When the transition is traversed, it is needed to ensure that the value of x has been assigned properly. Therefore, transition T, along with variable x that needs to be observed, is added into $\overline{\mathcal{A}}$. The exception is that when the action of T has been confirmed (i.e., $T \in A$), there is no need to add $T(x)$ into $\overline{A}_{\nu^{-1}x}$.

$\overline{\mathcal{S}}$ and $\mathcal{S}$: The same as the backward axiom, transition T is added to $\overline{S}$ when it is not in S.

Backward Condition Axiom: The effect of a transition with a conditional action is to transfer an EFSM from one state to another when a certain condition is matched. Therefore, a conditional transition affects only the control flow; no data are altered. Therefore, it is safe to assume that a conditional action is always correct, and leave the error detection to the state confirming path (which is handled by fields $\mathcal{S}$ and $\overline{\mathcal{S}}$). Hence, a conditional transition is always put into field $\mathcal{A}$.

In addition, if the boolean expression $\mathcal{C}$ contains any variables, those variables need to be defined and is put into $\overline{\mathcal{D}}$. If its tail state has not been confirmed, the transition is added to $\overline{\mathcal{S}}$.

Forward Condition Axiom: The forward condition axiom is similar to the backward axioms, except that those variables in $\mathcal{C}$ that have already in $\mathcal{D}$ do not need to be inserted into $\overline{\mathcal{D}}$.

Backward Input Axiom: Since an input statement can be viewed as an assignment statement that assigns input values to variables, their axiom is almost the same as the backward assignment axiom. The difference is that since an input statement is an external action, the observed event should be inserted into $\mathcal{E}$. The value κ in the axiom is an arbitrary input value.

Forward Input Axiom: The forward input axiom is similar to the backward input axiom, only that the input event is appended to the rear of the sequence in field $\mathcal{E}$ instead of to the front.

Backward Output Axiom: Since an output action is external, its output event is inserted into $\mathcal{E}$. Also, because the output of the transition automatically reveals the correctness of the action, transition T is included in $\mathcal{A}$. In addition, every variable contained in expression e must be included in $\overline{\mathcal{D}}$.

Forward Output Axiom: A forward output axiom is similar to the backward output axiom, except that the output event is appended after $\mathcal{E}$, and those variables that have been defined in $\mathcal{D}$ will not be inserted into $\overline{\mathcal{D}}$.

6.2 Test Case Generation

Different from the ATG method, in which the external events of a traversed path ensure only the correctness of the transition under test, the OAT method guarantees the correctness of all the transitions in the path. To test the correctness of an entire path, one must ensure that (1) every variable in the path contains a specific value, (2) the actions in the path produce expected effects, and (3) each transition in the path reaches its expected tail state. A path that satisfies these properties is called a *test path* and the sequence of external events generated by the test path is used as a test case. A test path can be found with the algorithm specified in Section 6.2.4, which can be divided into three major steps described in Sections 6.2.1, 6.2.2, and 6.2.3. Finally, an example of OAT method is described in Section 6.2.5.

6.2.1 Defining Variables

To guarantee that every variable in the generated test path contains a value, a given path is extended backward to form a new path, such that every variable in the new path is defined. Starting from the first state of a given path with a given assertion, backward axioms are applied to every possible incoming transition. Then, preconditions for these transitions are derived according to the axioms of the transitions' actions. If any of the derived assertion contains a $\overline{\mathcal{D}}$ field that contains no variables, the extended path will be the desired path. If no such assertion exists, the assertions just derived are used as post-conditions, and the new paths are extended one more transition backward. This is repeated until an assertion containing an empty $\overline{\mathcal{D}}$ field is found. This is a breadth first search backward through all possible paths to find one that defines every variable in the path. The procedure used in Figure 46 is similar to Procedure **Forward-Search** on page 100, only that Procedure **Defining-Variables** searches backward.

Note that it is possible that no such path exists. Two situations can happen. One is when the path is extended backward to a point that no more legal paths are available. In this case, the procedure signals that no path is found. The other case is the path being extended indefinitely, and still having some variable undefined. In this case, an upper bound is set to the maximum length under which a path can extend. If such length limitation is reached, one simply dictates that no such a path can be found.

6.2.2 Confirming Actions

To confirm the correctness of the actions in a given path, a set of rules, called *resolution rules*, is introduced. The rules move transitions from $\overline{\mathcal{A}}$ to $\mathcal{A}$ using the information provided by $\mathcal{E}$ and $\mathcal{P}$. A transition T with a variable set V in $\overline{\mathcal{A}}$ can be moved from $\overline{\mathcal{A}}$ to $\mathcal{A}$ if every variable x in V satisfies the following:

R1: x can be observed directly from an output event in $\mathcal{E}$.

Procedure Defining-Variables

description:
This procedure search backward through the state diagram of an EFSM to find a path that defines every variables in the path

input:
M: an EFSM
P: a path,
A: an assertion, and

output:
an extended path and
the assertion of the path's tail state

```
Put pair (P, A) in a queue Q
while Q is not empty do
    Get a pair (P', A') from Q
    Let s be the head state of P'
    for every incoming transition t of s do
        Apply appropriate axiom of t to A'
        Let A' the resulting assertion, and
        Append t the front of P'
        if field D̄ in A' is empty then
            return P' and A'
        end-if
        if P' is too long then
            Put (P', A') into Q
        end-if
    end-for
end-while
return path not found
```

Figure 46: Procedure **Defining-Variables**

R2: $\exists(y = e) \in \mathcal{P}$, such that $x \in \text{var}(e)$ and y satisfies resolution rules R1 or R2.

That is, transition T can be confirmed if the values of the variables in V either (1) appear in some output actions, or (2) propagate to other variables that can be observed from an output event.

With a given path and a given assertion, a new path that confirms all its actions can be found by a breadth-first search from the tail state of the given path. The procedure **Forward-Search**on page 100 is used to find a path whose associated assertion has an empty $\overline{\mathcal{A}}$ field. Note that, similar to the backward evaluation described in Section 6.2.1, a path that confirms all the actions may not exist (either no legal path is available or the length of the path exceeds a certain limit).

6.2.3 Distinguishing States

To ensure every transition in a path reaches a correct tail state, the same method used in Section 5.4.2 is now used here. The difference is that ATG only considers the mutants for the transition under test, while in OAT, all the mutants for those transitions in field $\overline{\mathcal{S}}$ must be generated.

When the mutants for a transition can be differentiated by the external events generated by the path, the transition is moved from field $\overline{\mathcal{S}}$ to $\mathcal{S}$. If the current path cannot differentiate the mutants, the path is extended forward using the forward axioms, and the newly generated external events are once again imposed to the mutants. This is repeated until an assertion with an empty $\overline{\mathcal{S}}$ is formed. Note that it is also possible that no such path can ever be found. An upper bound of the path length is also needed here as well as in Sections 6.2.1 and 6.2.2.

6.2.4 Generating a Test Case

Having the tools described in Sections 6.2.1, 6.2.2 and 6.2.3, the test case generation algorithm becomes trivial. First, an arbitrary transition is selected as a *seed transition*. Then, starting from the tail state of the seed transition and an assertion with empty fields, the corresponding backward axiom is applied to the seed transition. Using the seed transition and the resulting assertion as the initial path and assertion, the following three conditions are checked. (1) If the assertion contains a nonempty $\overline{\mathcal{D}}$ field, the path is extended backward with the procedure in Section 6.2.1, so that every variable in the path can be defined. (2) Otherwise, if $\overline{\mathcal{A}}$ is nonempty, the method in Section 6.2.2 is used to extend the path forward, so that every encountered action can be confirmed. (3) Otherwise, if $\overline{\mathcal{S}}$ is not empty, one can apply the method in Section 6.2.3 to extend the path forward, so that all the mutants can be differentiated. The path is repeatedly extended until $\overline{\mathcal{D}}$, $\overline{\mathcal{A}}$, and $\overline{\mathcal{S}}$ become empty. If any of the above three steps cannot find a suitable path, the EFSM is dictated to be untestable.

A minor adjustment must be made to the version numbers of the variables in an assertion when OAT changes its evaluation directions. When OAT changes from searching backward to searching forward, all the positive version numbers must be changed to negative values, and when it changes form searching forward to searching backward, the version numbers are changed to positive values. That is, if the highest version of variable x in a backward evaluation is i (i.e., $\nu^i x$ exists in the assertion), then R is changed into $R_{\nu^{-i}x}$. On the other hand, if the assertion is from a forward evaluation and lowest version of x is $-i$, R is changed into $R_{\nu^i x}$

6.2.5 An Example

Figures 47 to 49 contain an example of how a test case can be developed using transition 4 in Figure 6 as a seed transition. In Figure 47a, starting from the left-hand side assertion (assertion 1, in which every field is empty), one can derive the right-hand side assertion by applying the input backward axiom to assertion 1. Since $\overline{\mathcal{D}}$ of the resulting assertion is empty, the algorithm in Section 6.2.2 is used to confirm the action "$R?A$." First, the version numbers of assertion 2 are converted to forward axiom style, which is shown in assertion 3. Then, starting from assertion 3 in Figure 47b, path $\langle 6,7,2,3\rangle$ that empties field $\overline{\mathcal{A}}$ in assertion 7 is found. By applying resolution rules R1 and R2, the last assertion (assertion 9) is derived. The new assertion contains a nonempty $\overline{\mathcal{D}}$ field, so the method in Section 6.2.1 is resumed. After converting the version numbers to positive values, and applying backward axioms to assertion 10, assertion 13 is obtained in Figure 48a. The assertion has a $\overline{\mathcal{A}}$ field that contains transitions 2 and 1; therefore, the procedure in Section 6.2.2 is resumed to confirm the actions of the transitions. Fortunately, this can be resolved by the resolution rule, so no extra path is explored (Figure 48b). Now, both $\overline{\mathcal{D}}$ and $\overline{\mathcal{A}}$ are empty, so the process in Section 6.2.3 is performed. First, mutations for transitions 2, 3, 7, 1, and 6 can be differentiated by the path already derived (assertion 16 to assertion 17). Only the mutants of transition 4 in Figure 33 can mimic the specification. Therefore, additional path $\langle 4,5,3\rangle$ is explored. This results in assertion 18 in Figure 49. The additional path differentiates the mutant for transitions 4 and 5, and the final assertion (assertion 19) is derived.

Now, fields $\overline{\mathcal{D}}$, $\overline{\mathcal{A}}$ and $\overline{\mathcal{S}}$ are all empty, and the algorithm stops. The resulting path, $\langle 1, 2, 3, 4, 6, 7, 2, 3, 4, 5, 3 \rangle$ is the desired test path and field $\mathcal{E}$ in the last assertion contains the test case for transitions 1, 2, 3, 4, 5, 6, and 7. The values of the variables in $\mathcal{E}$ can be found in field $\mathcal{P}$. For example, the first external event shown in the test case is $U?\nu^{-1}M$, which suggests sending an input, $\nu^{-1}M$, to gate U. The value of $\nu^{-1}M$ specified in $\mathcal{P}$ field is κ_2, an arbitrary input value. Second, an output that contains the message $\nu^{-1}M$ (whose value is κ_2) and a sequence number $\nu^{-1}B$ (whose value is 0) is expected from gate R. The rest of the test case can be described in the similar manner.

6.3 Test Suite Generation

A test path generated by OAT may not contain all the transition in an EFSM. If this is the case, one can simply select a transition that is not included in the test path as a seed transition and generate another test case from it. This is repeated until every transition in the EFSM is covered by some test paths.

When the first path is constructed, the fields in the initial assertion are set to be all empty. However, to construct the second or the later test cases, one does not need to start from scratch. The transition that has been confirmed can be included in the initial assertion; i.e., fields $\mathcal{A}$ and $\mathcal{S}$ from the last test case can be used to start the current test case.

Those test cases generated need to be connected into a test suite. Procedure **Link** on page 113 can be used for such a purpose.

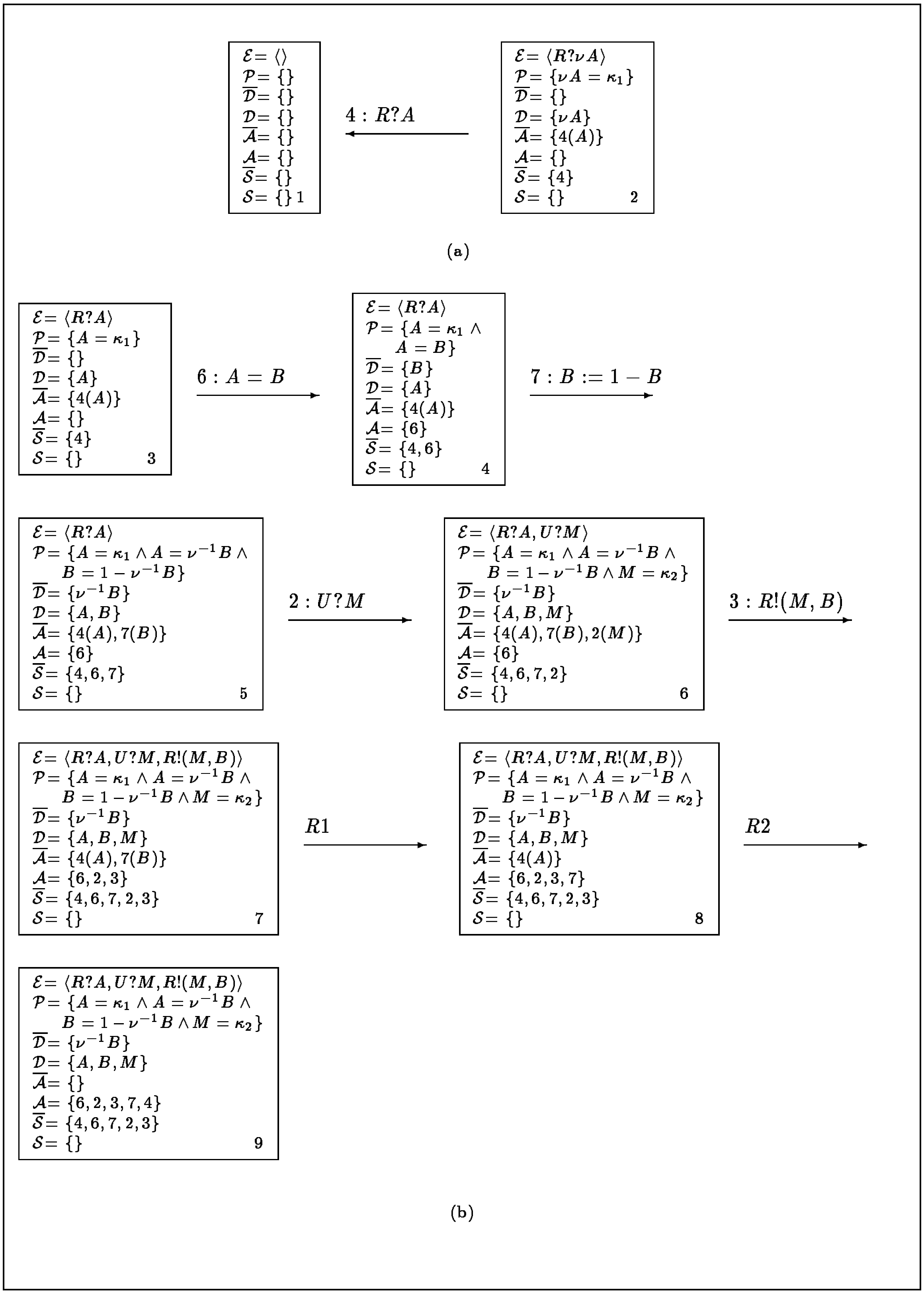


Figure 47: Test case generated from seed transition 4

$\mathcal{E}= \langle R?\nu A, U?\nu M, R!(\nu M, \nu B)\rangle$
$\mathcal{P}= \{\nu A = \kappa_1 \wedge \nu A = B \wedge \nu B = 1 - B \wedge \nu M = \kappa_2\}$
$\overline{\mathcal{D}}= \{B\}$
$\underline{\mathcal{D}}= \{\nu A, \nu B, \nu M\}$
$\overline{\mathcal{A}}= \{\}$
$\underline{\mathcal{A}}= \{6, 2, 3, 7, 4\}$
$\overline{\mathcal{S}}= \{4, 6, 7, 2, 3\}$
$\mathcal{S}= \{\}$
10

$\xleftarrow{3\,:\,R!(M,\,B)}$

$\mathcal{E}= \langle R!(M, B), R?\nu A, U?\nu M, R!(\nu M, \nu B)\rangle$
$\mathcal{P}= \{\nu A = \kappa_1 \wedge \nu A = B \wedge \nu B = 1 - B \wedge \nu M = \kappa_2\}$
$\overline{\mathcal{D}}= \{B, M\}$
$\underline{\mathcal{D}}= \{\nu A, \nu B, \nu M\}$
$\overline{\mathcal{A}}= \{\}$
$\underline{\mathcal{A}}= \{6, 2, 3, 7, 4\}$
$\overline{\mathcal{S}}= \{4, 6, 7, 2, 3\}$
$\mathcal{S}= \{\}$
11

$\xleftarrow{2\,:\,U?M}$

$\mathcal{E}= \langle U?M, R!(M, B), R?\nu A, U?\nu M, R!(\nu M, \nu B)\rangle$
$\mathcal{P}= \{\nu A = \kappa_1 \wedge \nu A = B \wedge \nu B = 1 - B \wedge \nu M = \kappa_2 \wedge M = \kappa_3\}$
$\overline{\mathcal{D}}= \{B\}$
$\underline{\mathcal{D}}= \{\nu A, \nu B, \nu M, M\}$
$\overline{\mathcal{A}}= \{\}$
$\underline{\mathcal{A}}= \{6, 2, 3, 7, 4\}$
$\overline{\mathcal{S}}= \{4, 6, 7, 2, 3\}$
$\mathcal{S}= \{\}$
12

$\xleftarrow{1\,:\,B\,:=\,0}$

$\mathcal{E}= \langle U?M, R!(M, \nu B), R?\nu A, U?\nu M, R!(\nu M, \nu^2 B)\rangle$
$\mathcal{P}= \{\nu A = \kappa_1 \wedge \nu A = \nu B \wedge \nu^2 B = 1 - \nu B \wedge \nu M = \kappa_2 \wedge M = \kappa_3 \wedge \nu B = 0\}$
$\overline{\mathcal{D}}= \{\}$
$\underline{\mathcal{D}}= \{\nu A, \nu^2 B, \nu M, M, \nu B\}$
$\overline{\mathcal{A}}= \{1(\nu B)\}$
$\underline{\mathcal{A}}= \{6, 2, 3, 7, 4\}$
$\overline{\mathcal{S}}= \{4, 6, 7, 2, 3, 1\}$
$\mathcal{S}= \{\}$
13

(a)

$\mathcal{E}= \langle U?\nu^{-1}M, R!(\nu^{-1}M, \nu^{-1}B), R?A, U?M, R!(M, B)\rangle$
$\mathcal{P}= \{A = \kappa_1 \wedge A = \nu^{-1}B \wedge B = 1 - \nu^{-1}B \wedge M = \kappa_2 \wedge \nu^{-1}M = \kappa_3 \wedge \nu^{-1}B = 0\}$
$\overline{\mathcal{D}}= \{\}$
$\underline{\mathcal{D}}= \{A, B, M, \nu^{-1}M, \nu^{-1}B\}$
$\overline{\mathcal{A}}= \{2(\nu^{-1}M), 1(\nu^{-1}B)\}$
$\underline{\mathcal{A}}= \{6, 2, 3, 7, 4\}$
$\overline{\mathcal{S}}= \{4, 6, 7, 2, 3, 1\}$
$\mathcal{S}= \{\}$
14

$\xrightarrow{R2}$

$\mathcal{E}= \langle U?\nu^{-1}M, R!(\nu^{-1}M, \nu^{-1}B), R?A, U?M, R!(M, B)\rangle$
$\mathcal{P}= \{A = \kappa_1 \wedge A = \nu^{-1}B \wedge B = 1 - \nu^{-1}B \wedge M = \kappa_2 \wedge \nu^{-1}M = \kappa_3 \wedge \nu^{-1}B = 0\}$
$\overline{\mathcal{D}}= \{\}$
$\underline{\mathcal{D}}= \{A, B, M, \nu^{-1}M, \nu^{-1}B\}$
$\overline{\mathcal{A}}= \{\}$
$\underline{\mathcal{A}}= \{6, 2, 3, 7, 4, 1\}$
$\overline{\mathcal{S}}= \{4, 6, 7, 2, 3, 1\}$
$\mathcal{S}= \{\}$
15

(b)

Figure 48: Test case generated from seed transition 4 (continued)

$\mathcal{E} = \langle U?\nu^{-1}M, R!(\nu^{-1}M, \nu^{-1}B),$
$\quad R?A, U?M, R!(M,B)\rangle$
$\mathcal{P} = \{A = \kappa_1 \wedge A = \nu^{-1}B \wedge$
$\quad B = 1 - \nu^{-1}B \wedge M = \kappa_2 \wedge$
$\quad \nu^{-1}M = \kappa_3 \wedge \nu^{-1}B = 0\}$
$\overline{\mathcal{D}} = \{\}$
$\mathcal{D} = \{A, B, M, \nu^{-1}M, \nu^{-1}B\}$
$\overline{\mathcal{A}} = \{\}$
$\mathcal{A} = \{6, 2, 3, 7, 4, 1\}$
$\overline{\mathcal{S}} = \{4, 6, 7, 2, 3, 1\}$
$\mathcal{S} = \{\}$ 16

$\longrightarrow$

$\mathcal{E} = \langle U?\nu^{-1}M, R!(\nu^{-1}M, \nu^{-1}B),$
$\quad R?A, U?M, R!(M,B)\rangle$
$\mathcal{P} = \{A = \kappa_1 \wedge A = \nu^{-1}B \wedge$
$\quad B = 1 - \nu^{-1}B \wedge M = \kappa_2 \wedge$
$\quad \nu^{-1}M = \kappa_3 \wedge \nu^{-1}B = 0\}$
$\overline{\mathcal{D}} = \{\}$
$\mathcal{D} = \{A, B, M, \nu^{-1}M, \nu^{-1}B\}$
$\overline{\mathcal{A}} = \{\}$
$\mathcal{A} = \{6, 2, 3, 7, 4, 1\}$
$\overline{\mathcal{S}} = \{4\}$
$\mathcal{S} = \{2, 3, 7, 1, 6\}$ 17

$\xrightarrow{4\,:\,R?A}$ $\xrightarrow{5\,:\,A <> B}$ $\xrightarrow{3\,:\,R!(M,B)}$

$\mathcal{E} = \langle U?\nu^{-1}M, R!(\nu^{-1}M, \nu^{-1}B),$
$\quad R?\nu^{-1}A, U?M, R!(M,B), R?A,$
$\quad R!(M,B)\rangle$
$\mathcal{P} = \{\nu^{-1}A = \kappa_1 \wedge \nu^{-1}A = \nu^{-1}B \wedge$
$\quad B = 1 - \nu^{-1}B \wedge M = \kappa_2 \wedge \nu^{-1}M = \kappa_3 \wedge$
$\quad \nu^{-1}B = 0 \wedge A = \kappa_4 \wedge A \neq B\}$
$\overline{\mathcal{D}} = \{\}$
$\mathcal{D} = \{\nu^{-1}A, B, M, \nu^{-1}M, \nu^{-1}B\}$
$\overline{\mathcal{A}} = \{\}$
$\mathcal{A} = \{6, 2, 3, 7, 4, 1, 5\}$
$\overline{\mathcal{S}} = \{4, 5\}$
$\mathcal{S} = \{2, 3, 7, 1, 6\}$ 18

$\longrightarrow$

$\mathcal{E} = \langle U?\nu^{-1}M, R!(\nu^{-1}M, \nu^{-1}B),$
$\quad R?\nu^{-1}A, U?M, R!(M,B), R?A,$
$\quad R!(M,B)\rangle$
$\mathcal{P} = \{\nu^{-1}A = \kappa_1 \wedge \nu^{-1}A = \nu^{-1}B \wedge$
$\quad B = 1 - \nu^{-1}B \wedge M = \kappa_2 \wedge \nu^{-1}M = \kappa_3 \wedge$
$\quad \nu^{-1}B = 0 \wedge A = \kappa_4 \wedge A \neq B\}$
$\overline{\mathcal{D}} = \{\}$
$\mathcal{D} = \{\nu^{-1}A, B, M, \nu^{-1}M, \nu^{-1}B\}$
$\overline{\mathcal{A}} = \{\}$
$\mathcal{A} = \{6, 2, 3, 7, 4, 1, 5\}$
$\overline{\mathcal{S}} = \{\}$
$\mathcal{S} = \{2, 3, 7, 1, 6, 4, 5\}$ 19

Figure 49: Test case generated from seed transition 4 (continued)

6.4 Summary

An axiomatic TCG method that optimize the overall length of a test suite is proposed in this chapter. The method uses more complicated assertions and axioms and utilizes both forward and backward evaluations.

Unlike ATG, several transitions are considered simultaneously in OAT. A test case is generated by searching backward to find a path that defines every variable in the path, and by searching forward for a path that confirms the actions and the tail states of the transitions in the path. These steps are repeated until every variable in the final test path is defined and every transition in the path contains confirmed action and reaches confirmed tail state.

Both ATG and OAT are used to generate test cases for single-transition faults. Although a TCG method designated for a certain fault model can detect other type of error, however, such detection is not guaranteed and cannot bring to the user any confidence of the protocol implementation's correctness. It will certainly be desirable to have a TCG method that generates test cases for any given fault models. The TCG method introduced in the next chapter is an attempt to fulfill this need.

CHAPTER VII

Test Case Generation for Given Fault Models (TGF)

7.1 Fault Models and TCG

A test case is meaningless if one does not know its purposes. The *test purposes* are statements showing what type of error the test case tries to detect. The errors are called *fault models*, which must be given before any meaningful test cases can be constructed. Unfortunately, all of the works mentioned in Chapters III and IV fail to state their fault models. Although their fault models can be implicitly realized by observing how their test cases are generated, the absence of explicitly stating so gives the impression that having fault models beforehand is unnecessary. It is unreasonable to have a test case generation method first, and then try to discover what errors it detects.

In this chapter, the test case generation problem is viewed from a different perspective. Instead of using fixed fault models to generate test cases, the definition of fault models is left to the TCG users. A general method is proposed such that test cases can be generated by comparing the differences between the specification and the given fault models.

TGF is a TCG method that takes a specification and a fault mode (both specified in EFSM) as its input. Then, the dynamic properties of the EFSMs are analyzed. By comparing the differences between the specification and the fault model, TGF is able to select a test case that detects errors described in the fault models.

7.2 The Behavior Model

Definition 2.3.1 in Section 2.3 clearly describes the structure of an EFSM. However, the definition is not sufficient to describe the *dynamic properties* of the EFSM. In this section, another model, called the *behavior model*, is proposed to describe such properties. A behavior model describes how the status of an EFSM changes while the EFSM is executed, and is specified by a *behavior function*, which will be described in detail in Section 7.2.3.

7.2.1 Scenarios

Recall that in Section 2.5, a version of a variable is defined as a different name for the same variable to store its different values. Formally, a version is an element in set $V \times \mathcal{Z}$, where V is a set of variables, and $\mathcal{Z}$ is the set of all integers. A *scenario* describes the relationship between versions and their values. That is, it stores the information of what version contains which value. In other words, a scenario records the "development" of the variables' values; it not only describes *what* value a variable stores, but also shows *when* it is stored. Formally, a scenario is a function that maps a set of versions to a set of values. It is denoted by a sequence of versions and their corresponding values quoted by a pair of angle brackets ($\langle\ \rangle$). For example, a scenario of the EFSM in Figure 6 might be $\langle B = 0, A = 0, \nu B = 1, \nu A = 0, ...\rangle$, which indicates that the values of variables B and A are zeros originally, and change to 1 and 0, respectively, later on.

7.2.2 Assertions and Axioms

In Section 2.5, an assertion is defined as a set of status. To be more precisely, an assertion should have been defined as a set of scenarios. Examining the assertions in Chapters V and VI, one can find out that the assertions actually describe the possible relationship between versions and values. For example, assertion $\{ \nu B = 1 \}$ contains both scenarios $\langle B = 1, \nu B = 1, \nu^2 B = 1, ...\rangle$ and $\langle B = 0, \nu B = 1, \nu^2 B = 0, ...\rangle$. Therefore, an axiom is a function that transforms one set of scenarios to another.

The assertions and axioms used in the behavior model is listed in Table 11. The predicates of the actions (column **Predicate**) state the condition that must be satisfied after the type of action is executed. The table also shows which variables are modified by the actions (column **New Versions**).

Table 11: The axioms of the action types in Figure 6

Type	Action	Predicate	New Versions
Assignment	$x := e$	$\nu x = e$	νx
Input	$G?x$	$\nu\tau = (G?\nu x) \wedge \nu x = \nu\kappa$	$\nu\tau, \nu x, \nu\kappa$
Output	$G!e$	$\nu\tau = (G!e)$	$\nu\tau$
Condition	$\mathcal{C}$	$\mathcal{C}$	none

In the first row of the table, an assignment action $x := e$ assigns the result of expression e to variable x. Since x will be modified after the assignment, it should be upgraded into a new version, which is denoted as νx (as shown in the **New Versions** column). Since, the new version of x will be equal to the result of e, any scenarios developed after the assignment action must satisfy $\nu x = e$, which is shown in the **Predicate** column. Similarly, the predicate of input action indicates that the next external event ($\nu\tau$) is receiving a new version of x from gate G ($\nu\tau = (G?\nu x)$), and the new version of x is equal to an arbitrary input, denoted $\nu\kappa$ ($\nu x = \nu\kappa$). After the action, τ, x, and κ are upgraded. For convenience, κ, an arbitrary input constant, is treated as a variable, so that operator "ν" can be used to indicate different versions of input values. This avoids marking input variables as κ_1, κ_2, etc., as proposed in Chapters V and VI. The output action is similar to the input action and is shown in the third row of the table. Recall that τ is an imaginary variable that record the current external event (see Definition 2.3.1 on page 19). Associating the operator "ν" with τ ($\nu\tau$) indicates the next external event. In the fourth row of Table 6, if the condition is a boolean expression $\mathcal{C}$, any scenarios developed after the condition action must satisfy $\mathcal{C}$. Since no variables are changed in the action, no variable is upgraded.

7.2.3 The Behavior Function

Any transition T pointing from state h to state t_T imposes a restriction on whatever scenarios that can be developed after h. Let A_T be the assertion that indicates the restriction imposed by T, and let V_T be the set of new versions created by T. If the set of scenarios developed after state h through transition T is denoted as $\beta_T[h]$, then

$\beta_T[h] = A_T \cap \beta[t_T]_{V_T}$, meaning that the scenarios developed after h through T are those developed after t_T (with upgraded versions) that satisfy A_T.

If a state h has more than one outgoing transition, T_1, T_2, ..., T_n (Figure 50a), the *parallel composition* of the behavior at state h is defined as follows.

Definition 7.2.1

$$\begin{aligned} \beta[h] &= \beta_{T_1}[h] \cup \beta_{T_2}[h] \cup \cdots \cup \beta_{T_n}[h] \\ &= (A_{T_1} \cap \beta[t_{T_1}]_{V_{T_1}}) \cup (A_{T_2} \cap \beta[t_{T_2}]_{V_{T_2}}) \cup \cdots \cup (A_{T_n} \cap \beta[t_{T_n}]_{V_{T_n}}) \end{aligned} \tag{7.1}$$

The *serial composition* of $\beta_p[h]$ along a certain path $p = \langle T_1, T_2, ..., T_n \rangle$ (Figure 50b), is a derivation of the behavior of the states and transitions along the path. That is,

$$\begin{aligned} \beta_p[h] &= A_{T_1} \cap \beta_{\langle T_2, T_3, ..., T_n \rangle}[t_{T_1}]_{V_{T_1}} \\ &= A_{T_1} \cap (A_{T_2} \cap \beta_{\langle T_3, T_4, ... T_n \rangle}[t_{T_2}]_{V_{T_2}})_{V_{T_1}} \\ &= A_{T_1} \cap (A_{T_2})_{V_{T_1}} \cap \beta_{\langle T_3, T_4, ..., T_n \rangle}[t_{T_2}]_{V_{T_1} \uplus V_{T_2}} \\ &\vdots \\ &= A_{T_1} \cap (A_{T_2})_{V_{T_1}} \cap (A_{T_3})_{V_{T_1} \uplus V_{T_2}} \cap \cdots \cap \beta[t_{T_n}]_{V_{T_1} \uplus V_{T_2} \uplus \cdots \uplus V_{T_n}} \end{aligned} \tag{7.2}$$

where the operator "$\uplus$" is defined as follows:

Definition 7.2.2 *Let V_1 and V_2 be two sets of versions. Then*

$$\begin{aligned} V_1 \uplus V_2 = \quad & \{\nu^{i+j}w \mid \exists \nu^i w \in V_1 \wedge \exists \nu^j w \in V_2\} \\ \cup \quad & \{\nu^i w \mid \exists \nu^i w \in V_1 \wedge \forall \nu^j w \notin V_2\} \\ \cup \quad & \{\nu^j w \mid \forall \nu^i w \notin V_1 \wedge \exists \nu^j w \in V_2\} \end{aligned} \tag{7.3}$$

In other words, the result of $V_1 \uplus V_2$ is a version set that contains those variables upgraded by V_1 and then by V_2 subsequently.

In Figure 50a, any transition in T_1 through T_n can be executed next. Therefore, $\beta[h]$ is the union of all possible scenarios derived through the execution of every

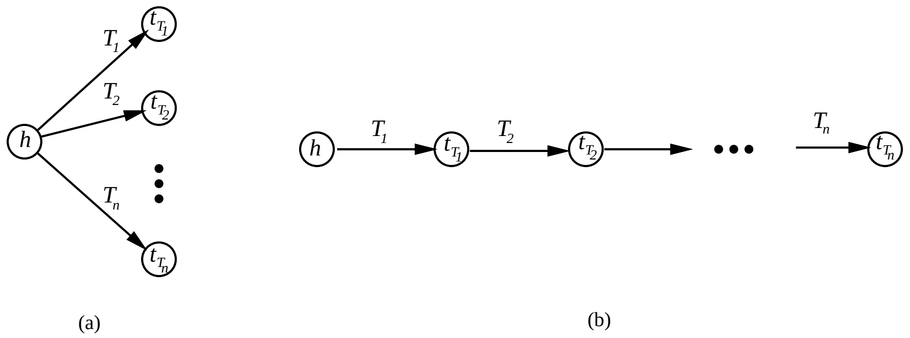


Figure 50: (a) Parallel and (b) serial composition of behaviors

outgoing transition. On the other hand, in Figure 50b, $\beta_p[h]$ must contain those scenarios that satisfy the condition set by every transition in the sequence. Thus, it is the intersection of all the assertions provided by the transitions. A formal definition of the behavior model is shown in the following definition.

Definition 7.2.3 *The behavior of an EFSM is a sextuple, $\langle Q, V, K, Z, \Pi, \beta \rangle$, where*

1. *Q is a set of states,*
2. *V is a set of variables,*
3. *K is a finite or infinite set of values,*
4. *$Z = V \times \mathcal{Z}$ is a set of versions (where $\mathcal{Z}$ represents nonnegative integers),*
5. *$\Pi = \{\pi \mid \pi : Z \to K\}$ is a set of scenarios, and*
6. *$\beta : Q \to 2^{\Pi}$ is a behavior function.*

The behavior function of the EFSM in Figure 6 can be defined in Figure 51. According to Definition 7.2.1 and the assignment axiom shown in Table 11, $\beta[a]$ is the intersection of those scenarios that satisfy the condition imposed by "$B := 0$" (which is $\nu B = 0$) and whatever scenarios developed after state b ($\beta[b]$), except that variable B is upgraded to a higher version (denoted $\beta[b]_{\nu B}$). Similarly, $\beta[e]$ is parallel composed from two outgoing transitions of state e, which includes the scenarios that traverse through either transition 6 or transition 5. Those scenarios traversed through transition 6 must satisfy $\{A = B\} \cap \beta[f]$, while others must satisfy $\{A \neq B\} \cap \beta[c]$.

$$\boxed{\begin{array}{l}
\beta[a] = \{\nu B = 0\} \cap \beta[b]_{\nu B} \\
\beta[b] = \{\nu\tau = (U?\nu M) \ \wedge \ \nu M = \nu\kappa\} \cap \beta[c]_{\nu\tau,\nu M,\nu\kappa} \\
\beta[c] = \{\nu\tau = (R!(M, B))\} \cap \beta[d]_{\nu\tau} \\
\beta[d] = \{\nu\tau = (R?\nu A) \ \wedge \ \nu A = \nu\kappa\} \cap \beta[e]_{\nu\tau,\nu A,\nu\kappa} \\
\beta[e] = (\{A = B\} \cap \beta[f]) \cup (\{A \neq B\} \cap \beta[c]) \\
\beta[f] = \{\nu B = 1 - B\} \cap \beta[b]_{\nu B}
\end{array}}$$

Figure 51: The behavior of the EFSM in Figure 6

Based on the recursive function in Figure 51, $\beta[a]$ can be derived as follows:

$$\begin{aligned}
\beta[a] &= \{\nu B = 0\} \cap \beta[b]_{\nu B} \\
&= \{\nu B = 0\} \\
&\quad \cap(\{\nu\tau = (U?\nu M) \wedge \nu M = \nu\kappa\} \\
&\quad \cap\beta[c]_{\nu B,\nu\tau,\nu M,\nu\kappa}) \\
&= \{\nu B = 0 \wedge \nu\tau = (U?\nu M) \wedge \nu M = \nu\kappa\} \\
&\quad \cap\beta[c]_{\nu B,\nu\tau,\nu M,\nu\kappa} \\
&= \{\nu B = 0 \wedge \nu\tau = (U?\nu M) \wedge \nu M = \nu\kappa \qquad\qquad (7.4)\\
&\quad \wedge\nu^2\tau = (R!(\nu M, \nu B))\} \cap \beta[d]_{\nu B,\nu^2\tau,\nu M,\nu\kappa} \\
&= \{\nu B = 0 \wedge \nu\tau = (U?\nu M) \wedge \nu M = \nu\kappa \\
&\quad \wedge\nu^2\tau = (R!(\nu M, \nu B)) \\
&\quad \wedge\nu^3\tau = (R?\nu A) \wedge \nu A = \nu^2\kappa\} \\
&\quad \cap\beta[e]_{\nu B,\nu^3\tau,\nu M,\nu^2\kappa,\nu A} \\
&= \ldots
\end{aligned}$$

The set of all possible scenarios that can be derived from an EFSM M is denoted by $\beta[\![M]\!]$. Formally, $\beta[\![M]\!] = \bigcup_{\forall s \in Q} \beta[s]$, where Q is the state set of M.

Note that unless every legal path in an EFSM ends up at a final state, the recursive function will never terminate. That is, the length of a scenario can be infinite. In Section 7.3.3, a method of how to obtain a finite external event sequence from the behavior function described above will be discussed.

7.3 Test Case Generation

7.3.1 Functionally Equivalent

Two scenarios are *functionally equivalent* if they generate the same external events. For example, scenarios $\langle \nu A = 0, \nu M = \nu\kappa, \nu B = 1, \nu\tau = (R!(\nu M, \nu B))\rangle$ and $\langle \nu A = 1, \nu M = \nu\kappa, \nu B = 1, \nu\tau = (R!(\nu^M, \nu B))\rangle$ are functionally equivalent since both output a "$(\nu\kappa, 1)$" through gate R. The value of νA is insignificant in the example since it does not affect the outcome. Formally, scenarios h and g are functionally equivalent if $h(\nu^i\tau) = g(\nu^i\tau)$ for any integer i. Recall that a scenario is a function that maps a set of versions to a set of values. Therefore, $h(\nu^i\tau)$ denotes the value of $\nu^i\tau$ in scenario h.

A *functionally equivalent set (FES)* of an assertion A is a set of all functionally equivalent scenarios of the scenarios in assertion A. The FES of A, denoted by $\mathbf{FES}(A)$, is defined as follows:

Definition 7.3.1 $\mathbf{FES}(A) = \{g \mid \forall i \geq 0,\ \exists h \in A, [h(\nu^i\tau) = g(\nu^i\tau)]\}$

Corollary 7.3.2

1. $\mathbf{FES}(A \cup B) = \mathbf{FES}(A) \cup \mathbf{FES}(B)$

2. $\mathbf{FES}(A \cap B) = \mathbf{FES}(A) \cap \mathbf{FES}(B)$

Proof.

$$
\begin{array}{lrcl}
1. & \mathbf{FES}(A \cup B) & = & \{g \mid \forall i \geq 0, \exists h \in A \cup B, [h(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & = & \{g \mid \forall i \geq 0, \exists h_A \in A, [h_A(\nu^i\tau) = g(\nu^i\tau)] \\
 & & & \vee \exists h_B \in B, [h_B(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & = & \{g \mid \forall i \geq 0, \exists h_A[h_A(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & & \cup \{g \mid \forall i \geq 0, \exists h_B \in B, [h_B(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & = & \mathbf{FES}(()A) \cup \mathbf{FES}(B) \\
2. & \mathbf{FES}(A \cap B) & = & \{g \mid \forall i \geq 0, \exists h \in A \cup B, [h(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & = & \{g \mid \forall i \geq 0, \exists h_A \in A, [h_A(\nu^i\tau) = g(\nu^i\tau)] \\
 & & & \wedge \exists h_B \in B, [h_B(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & = & \{g \mid \forall i \geq 0, \exists h_A[h_A(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & & \cap \{g \mid \forall i \geq 0, \exists h_B \in B, [h_B(\nu^i\tau) = g(\nu^i\tau)]\} \\
 & & = & \mathbf{FES}(()A) \cap \mathbf{FES}(B)
\end{array}
$$

□

7.3.2 Test Scenarios

A fault model can also be specified by a mutant. In this chapter, a mutant is not restricted in having only one erroneous transition. In fact, a mutant can be any EFSM, as long as it looks different from the specification. For example, Figure 52 is a mutant of the EFSM in Figure 6, in which the original transition 6 becomes five transitions that point to every state other than the correct state f. This particular type of mutant is useful for testing whether the original transition reaches the correct tail state (the transfer faults). The behavior function of Figure 52 is defined in

Figure 53. Note that because of the faulty transition 6, $\beta[e']$ is different from the $\beta[e]$ in Figure 51.

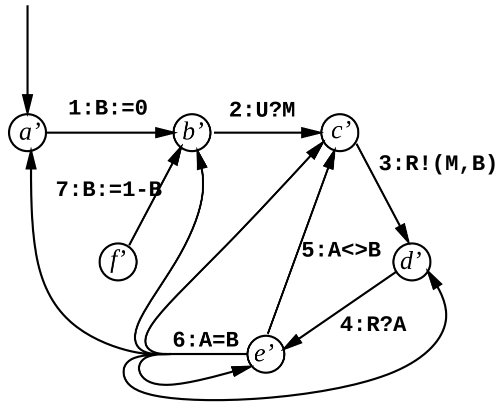


Figure 52: A mutant of the EFSM in Figure 6 with an erroneous transition 6

$$
\begin{aligned}
&\beta[a'] = \{\nu B = 0\} \cap \beta[b']_{\nu B} \\
&\beta[b'] = \{\nu\tau = (U?\nu M) \ \wedge \ \nu M = \nu\kappa\} \cap \beta[c']_{\nu\tau,\nu M,\nu\kappa} \\
&\beta[c'] = \{\nu\tau = (R!(M,B))\} \cap \beta[d']_{\nu\tau} \\
&\beta[d'] = \{\nu\tau = (R?\nu A) \ \wedge \ \nu A = \nu\kappa\} \cap \beta[e']_{\nu\tau,\nu A,\nu\kappa} \\
&\beta[e'] = (\{A = B\} \cap (\beta[a'] \cup \beta[b'] \cup \beta[c'] \cup \beta[d'] \cup \beta[e'])) \\
&\qquad\quad \cup(\{A \neq B\} \cap \beta[c']) \\
&\beta[f'] = \{\nu B = 1 - B\} \cap \beta[b']_{\nu B}
\end{aligned}
$$

Figure 53: The behavior of the EFSM in Figure 52

A *test scenario* is a scenario whose external event distinguishes the correct EFSM from the mutant ones. That is, the external event sequence generated by the test scenario cannot be reproduced by the mutants. A test scenario p that distinguishes a correct EFSM M from a mutant F can be defined as follows:

Definition 7.3.3 *A scenario p is a test scenario for mutant F if $p \in (\beta[\![M]\!] - \mathbf{FES}(\beta[\![F]\!]))$*

The meaning of Definition 7.3.3 is illustrated in Figure 54. There are three sets of scenarios in the figure. Sets 1 and 2 are all possible scenarios that can be developed by specification M and by its mutant F, respectively. Set 3 contains those scenarios that generate identical external events to some of the scenarios in set 2. A test scenario has to be developed by a correct implementation and should not be reproduced by any faulty implementations. Therefore, any scenario in the shaded area, which is $\beta[\![M]\!] - \mathbf{FES}(\beta[\![F]\!])$, can be used as a test scenario. To find a test scenario for a group of fault models, one can simply search for a scenario in $\beta[\![M]\!] - \cup_F \mathbf{FES}(\beta[\![F]\!])$.

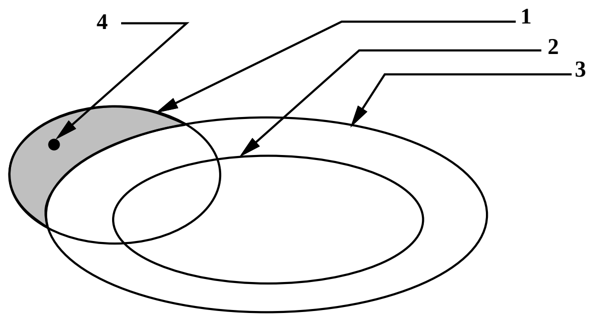


Figure 54: The relationship among $\beta[\![M]\!]$, $\beta[\![F]\!]$ and $\mathbf{FES}(\beta[\![F]\!])$: 1: The set of all scenarios that can be generated by M ($\beta[\![M]\!]$), 2: The set of all scenarios that can be generated by F ($\beta[\![F]\!]$), 3: All functionally equivalent scenarios of $\beta[\![F]\!]$ ($\mathbf{FES}(\beta[\![F]\!])$), and 4: A test scenario.

7.3.3 Test Cases

A *test case* in this chapter is formally defined as a prefix of the external event sequence generated by a test scenario. If h is a test scenario, the external event sequence generated by h is $\langle h(\tau), h(\nu\tau), h(\nu^2\tau), h(\nu^3\tau), ...\rangle$. Unfortunately, the sequence might contain an infinite number of elements. It is necessary to find a finite sequence that can distinguish a correct implementation from the erroneous ones. Therefore, a test case is defined as follows:

Definition 7.3.4 *Let h be a test scenario of an EFSM M for a mutant F. A sequence T is a test case if*

1. $T = \langle h(\tau), h(\nu\tau), ..., h(\nu^i\tau), ..., h(\nu^n\tau)\rangle$, *and*
2. $\forall g \in \mathbf{FES}(\beta[\![F]\!]), \exists i \leq n, [h(\nu^i\tau) \neq g(\nu^i\tau)]$.

That is, a test case is a finite external event sequence that contains at least one element that cannot be reproduced by a faulty implementation. Therefore, one can recognize an incorrect implementation once an unexpected external event is produced.

7.3.4 Test Case Generation Algorithms

For any specification M and its mutant F, it can be shown that $\beta[s] - \mathbf{FES}(\beta[\![F]\!]) \subseteq \beta[\![M]\!] - \mathbf{FES}(\beta[\![F]\!])$ (because $\beta[s] \subseteq \beta[\![M]\!]$, where s is a state in M). Though a test scenario can be found anywhere in $\beta[\![M]\!] - \mathbf{FES}(\beta[\![F]\!])$, it is more convenient to select the initial state as state s and to search for a test scenario in $\beta[s] - \mathbf{FES}(\beta[\![F]\!])$. Finding a test scenario from the latter not only prevents one from having to set up the IUT to the starting state of the test scenario, but also reveals the correctness of the initial transition. Theorem 7.3.7 provides a hint of finding $\beta[s] - \mathbf{FES}(\beta[\![F]\!])$.

Lemma 7.3.5 *Let h be a scenario and u be a state that the corresponding path of h passes through. Then, $h \in A \cap \beta[u]_V$, for some assertion A and version set V.*

Proof.

Basis: When u is the head state of the path, let $A = \{true\}$ and $V = \emptyset$. The lemma trivially holds.

Hypothesis: Assume that the lemma holds for a state v in the path; i.e., $h \in A_v \cap \beta[v]_{V_v}$, for some assertion A_v and version set V_v.

Induction: Let u be the next state of v. According to the serial composition rule,

$$\begin{aligned} h &\in A_v \cap \beta[v]_{V_v} \\ &= A_v \cap (A_u \cap \beta[u]_{V_v \uplus V_u}) \end{aligned} \tag{7.5}$$

where A_u is the condition imposed by the action of the transition from v to u, and V_u is the new version produced by the transition. Then, let $A = A_v \cap A_u$ and $V = V_v \uplus V_u$, the lemma holds.

□

Corollary 7.3.6 *Any set of scenarios, say H, can be represented as $\bigcup_h (A_h \cap \beta[u_h]_{V_h})$, where h is a scenario in H, and A_h, u_h, and V_h are an assertion, a state, and a set of new versions, respectively.*

Theorem 7.3.7 *Let H and G be two sets of scenarios of EFSM M_h and M_g, respectively, such that $H = \bigcup_h (A_h \cap \beta[u_h]_{V_h})$ and $G = \bigcup_g (A_g \cap \beta[u_g]_{V_g})$. Then,*

$$H - \mathbf{FES}(G) \supseteq \bigcup_h \bigcap_g K_{h,g} \tag{7.6}$$

where $K_{h,g}$ is the following formulas.

$$\begin{cases} A_h \cap \beta[u_h]_{V_h} & \text{if } A_h \cap \mathbf{FES}(A_g) = \emptyset \quad \text{(1a)} \\ (A_h - \mathbf{FES}(A_g)) \cap \beta[u_h]_{V_h} & \text{if } A_h \cap \mathbf{FES}(A_g) \neq \emptyset \\ & \text{and } A_h - \mathbf{FES}(A_g) \neq \emptyset \quad \text{(1b)} \\ A_h \cap (\beta[u_h]_{V_h} - \mathbf{FES}(\beta[u_g]_{V_g})) & \text{otherwise} \quad \text{(2)} \end{cases}$$

Proof.

$$\begin{aligned} & H - \mathbf{FES}(G) \\ = \; & \textstyle\bigcup_h (A_h \cap \beta[u_h]_{V_h}) - \mathbf{FES}(\bigcup_g (A_g \cap \beta[u_g]_{V_g})) \\ = \; & \textstyle\bigcup_h \bigcap_g [((A_h - \mathbf{FES}(A_g)) \cap \beta[u_h]_{V_h}) \cup (A_h \cap (\beta[u_h]_{V_h} - \beta[u_g]_{V_g}))] \\ \supseteq \; & \textstyle\bigcup_h \bigcap_g K_{h,g} \end{aligned}$$

$$\text{where} \quad K_{h,g} = \begin{cases} (A_h - \mathbf{FES}(A_g)) \cap \beta[u_h]_{V_h} & \text{if } A_h - \mathbf{FES}(A_g) \neq \emptyset \quad (1) \\ A_h \cap (\beta[u_h]_{V_h} - \mathbf{FES}(\beta[u_g]_{V_g})) & \text{otherwise} \quad (2) \end{cases}$$

$$= \begin{cases} A_h \cap \beta[u_h]_{V_h} & \text{if } A_h \cap \mathbf{FES}(A_g) = \emptyset \quad \text{(1a)} \\ (A_h - \mathbf{FES}(A_g)) \cap \beta[u_h]_{V_h} & \text{if } A_h \cap \mathbf{FES}(A_g) \neq \emptyset \\ & \text{and } A_h - \mathbf{FES}(A_g) \neq \emptyset \quad \text{(1b)} \\ A_h \cap (\beta[u_h]_{V_h} - \mathbf{FES}(\beta[u_g]_{V_g})) & \text{otherwise} \quad \text{(2)} \end{cases}$$

Note that equations (1a) and (1b) are derived from equation (1). □

As shown in the equations above, $K_{h,g}$ can be divided into three cases, Cases 1a, 1b, and 2, which are illustrated in Figure 55. Each circle in the figure represents a set of scenarios. In Case 2 (Figure 55a), A_h is covered by $\mathbf{FES}(A_g)$, meaning that for any scenario in A_h, there is a scenario in A_g that generates the same external event sequence as A_h does. Therefore, A_h along cannot distinguish H from G. To find the differences between H and G, a scenario that distinguishes $\beta[u_h]_{V_h}$ from $\mathbf{FES}(\beta[u_g]_{V_g})$ is needed. Therefore, $\beta[u_h]_{V_h} - \mathbf{FES}(\beta[u_g]_{V_g})$ is recursively calculated.

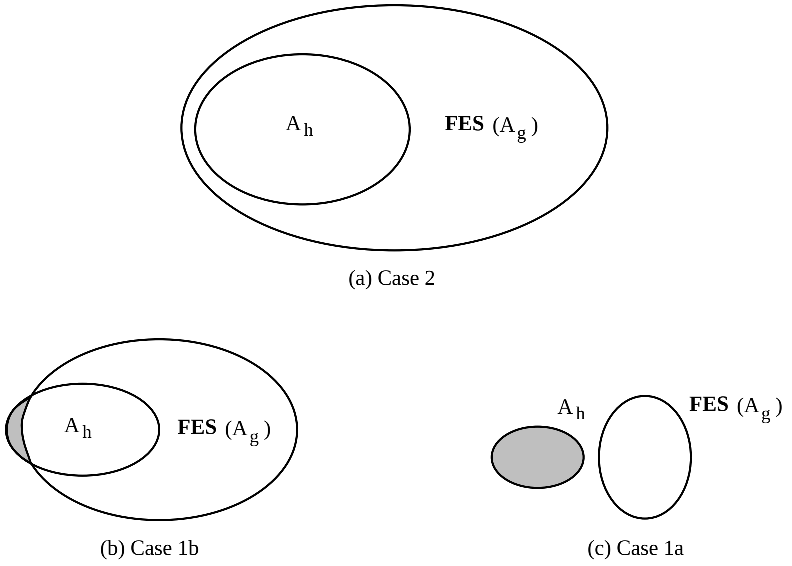


Figure 55: Illustration of the three cases of $K_{h,g}$

In Case 1b (Figure 55b), some scenarios in A_h are not covered by $\mathbf{FES}(A_g)$, which means that a scenario that distinguishes H from G can be found in A_h. Therefore, $\beta[u_h]_{V_h}$ is expanded to impose more restrictions such that a scenario in the shaded area of Figure 55b can be found.

In Case 1a (Figure 55c), no scenarios in A_h are covered by $\mathbf{FES}(A_g)$, which means every scenario in A_h can be used to distinguish H from G. Therefore, the external events generated by a scenario in A_h can be used as a test case.

Let $\beta[s]$ be H, and $\beta[\![F]\!]$ be G. Using the result of the equations above, $\beta[s] - \mathbf{FES}(\beta[\![F]\!])$ can be computed by Procedure **TGF** in Figure 56.

Procedure TGF

description:
Using Theorem 7.3.7 to find a test scenario in $\beta[s] - \mathbf{FES}(\beta[\![F]\!])$.

input:
M: A specification
F: A fault model

output:
a test path

```
Let there be a quintuple ⟨u, p, u′, p′, c⟩,
where u and u′ are assertions, p and p′ are states,
and c is respectively 1a, 1b, or 2 w.r.t. Cases 1a, 1b, or 2 above.
Let Q be a queue that initially contains ⟨s, true, u′, true, 2⟩ for every state u′ in F.
while Q is not empty do
    Get an element ⟨u, p, u′, p′, c⟩ from Q
    if c is in Case 1a then
        if there is an element ⟨v, q, v′, q′, d⟩ in Q and p = q then
            return p
        end-if
    else if c is in Case 1b then
        for every outgoing transition t from u do
            Put ⟨v_t, (p ∧ p_t), u′, p′, d⟩ into Q,
            where v_t is the state t pointing to,
            p_t is the assertion imposed by the action of t,
            and d is the relation between {p ∧ p_t} and FES({p′}).
        end-for
    else
        {c is in case 2}
        for every outgoing transition t of u, and t′ of u′ do
            put ⟨v_t, (p ∧ p_t), v′_t, (p′ ∧ p′_t), d⟩ into Q,
            where v_t and v′_t are tail states of t and t′, respectively
            p_t and p′_t assertions of t and t′,
            and d is the relation between {p ∧ p_t} and FES({p′ ∧ p′_t})
        end-for
    end-if
end-while
```

Figure 56: Procedure **TGF**

The idea of the Procedure can be illustrated by Figure 55. Initially, every element in Q belongs to Case 2, which is Figure 55a. In order to find a test case, $\beta[u]$ and $\beta[u']$ are expanded. This imposes more restrictions and less scenarios will satisfy the new restrictions, which makes the circles in Figure 55a "shrink." Eventually, if a test scenario exists, the figure will become either Figure 55b or Figure 55c. If it is Figure 55c, a test scenario is found. Otherwise, the algorithm simply expand $\beta[u]$, the behavior of the correct machine, such that the A_h circle in Figure 55b shrinks toward the shaded area and becomes Figure 55c.

7.3.5 An Example

Table 12 illustrates how a test case that distinguishes protocol specification in Figure 6 (denoted M) from its mutant in Figure 52 (denoted F) can be generated by the above algorithm. The table shows the content of the queue. Initially, the queue contains those elements listed in column 1 (Initial). Since all the five elements belongs to Case 3, both predicates for M and F are expanded according to Step 2c in the above algorithm. For example, the element in column 2 (1st Expansion) and row 1 is generated by the element in column 1 and row 1, where M traverses to state b and F traverses to state b'. Both assertions for M and F become A_1. Similarly, the elements in column 2 and rows 5–10 are generated by the element in column 1 and row 5. Since all the elements are in Case 3, each of them is expanded by Step 2c of the algorithm, which result in the elements in column 3 (2nd Expansion). Those elements that belong to Case 1 need not be expanded. The rest of the elements will be expanded according to Step 2c of the algorithm, and the resulting queue is shown in column 4 (3rd Expansion). Rows 12, 13, 14, 15, and 16 in column 4 are generated by rows 1, 6, 7, 11, and 10 in column 3, respectively.

Table 12: The illustration of how the test case of Figure 52 is generated

Row	Initial	1st Expansion	2nd Expansion
1	$\langle a, A_0, a', A_0, 2\rangle$	$\langle b, A_1, b', A_1, 2\rangle$	$\langle c, A_1 \cap A_2, c', A_1 \cap A_2, 2\rangle$
2	$\langle a, A_0, b', A_0, 2\rangle$	$\langle b, A_1, c', A_2, 2\rangle$	$\langle c, A_1 \cap A_2, d', A_2 \cap A_8, 1a\rangle$
3	$\langle a, A_0, c', A_0, 2\rangle$	$\langle b, A_1, d', A_3, 2\rangle$	$\langle c, A_1 \cap A_2, e', A_3 \cap A_9, 1a\rangle$
4	$\langle a, A_0, d', A_0, 2\rangle$	$\langle b, A_1, e', A_4, 2\rangle$	...all in Case 1a
5	$\langle a, A_0, e', A_0, 2\rangle$	$\langle b, A_1, c', A_5, 2\rangle$	$\langle c, A_1 \cap A_2, d', A_5 \cap A_3, 1a\rangle$
6		$\langle b, A_1, a', A_6, 2\rangle$	$\langle c, A_1 \cap A_2, b', A_6 \cap A_1, 2\rangle$
7		$\langle b, A_1, b', A_6, 2\rangle$	$\langle c, A_1 \cap A_2, c', A_6 \cap A_2, 2\rangle$
8		$\langle b, A_1, c', A_6, 2\rangle$	$\langle c, A_1 \cap A_2, d', A_6 \cap A_3, 1a\rangle$
9		$\langle b, A_1, d', A_6, 2\rangle$	$\langle b, A_1 \cap A_2, e', A_6 \cap A_4, 1a\rangle$
10		$\langle b, A_1, e', A_6, 2\rangle$	...all in Case 2
11	$\langle a, A_0, f', A_0, 2\rangle$	$\langle b, A_1, b', A_7, 2\rangle$	$\langle c, A_1 \cap A_2, c', A_7 \cap A_2, 2\rangle$

Row	3rd Expansion
12	$\langle d, A_1 \cap A_2 \cap A_{10}, d', A_1 \cap A_2 \cap A_{10}, 2\rangle$
13	$\langle d, A_1 \cap A_2 \cap A_{10}, c', A_6 \cap A_1 \cap A_2, 2\rangle$
14	$\langle d, A_1 \cap A_2 \cap A_{10}, d', A_6 \cap A_2 \cap A_8, 2\rangle$
15	$\langle d, A_1 \cap A_2 \cap A_{10}, d', A_7 \cap A_2 \cap A_8, 2\rangle$
16	elements expanded from row 10

$A_0 = true$

$A_1 = \{\nu B = 0\}$

$A_2 = \{\nu\tau = (U?\nu M) \wedge \nu M = \nu\kappa\}$

$A_3 = \{\nu\tau = (R!(M, B))\}$

$A_4 = \{\nu\tau = (R?\nu A) \wedge \nu A = \nu\kappa\}$

$A_5 = \{A \neq B\}$

$A_6 = \{A = B\}$

$A_7 = \{\nu B = 1 - B\}$

$A_8 = \{\nu^2\tau = (R!(\nu M, B))\}$

$A_9 = \{\nu^2\tau = (R?\nu A) \wedge \nu A = \nu\kappa\}$

$A_{10} = \{\nu^2\tau = (R!(\nu M, \nu B))\}$

Such expansions are repeated until the eighth time, when there is an element $\langle u, p, u', p', 1\rangle$ with

$$\begin{aligned} p = \{&\nu B{=}0 \wedge \nu\tau{=}(U?\nu M) \wedge \nu M{=}\nu\kappa \wedge \nu^2\tau{=}(R!(\nu M, \nu B)) \\ &\wedge \nu^3\tau{=}(R?\nu A) \wedge \nu A{=}\nu^2\kappa \wedge \nu A{=}\nu B \wedge \nu^2 B{=}1 - \nu B \\ &\wedge \nu^4\tau{=}(U?\nu^2 M) \wedge \nu^2 M{=}\nu^3\kappa \wedge \nu^5\tau{=}(R!(\nu^2 M, \nu^2 B))\} \end{aligned} \tag{7.7}$$

and none of the elements in the queue has the same predicate at their second position. Therefore, the external events of the assertion's corresponding path can be used as a test case, which is

$\langle \nu\tau{=}(U?\nu\kappa), \nu^2\tau{=}(R!(\nu\kappa, 0)), \nu^3\tau{=}(R?0), \nu^4\tau{=}(U?\nu^3\kappa),\ \nu^5\tau{=}(R!(\nu^3\kappa, 1))\rangle$.

The test case means that (1) input an arbitrary message to gate U, (2) expect an output from gate R whose sequence number is 0 and message is equal to the input in (1), (3) input an acknowledgment 0 through gate R, (4) input another message through gate U, and (5) expect the same message in (4) being output through gate R with sequence number 1. It can be shown that such a sequence cannot be reproduced by the mutant implementation, F.

7.4 Summary

In this chapter, it is argued that fault models must be given before any meaningful test case can be constructed. Fault models provide a test case with a purpose, so one can realize why the test case is applied. The TCG method, TGF, proposed in this chapter takes a protocol specification and its fault models (all specified by EFSMs) as its input, and produces a test case that detects the errors in the fault models.

Leaving the duty of specifying fault models to the users provides more flexibility for generating test cases. Users can generate those test cases that detect those faults they think are most important. When the budget is tight and the time is limited, being able to test those critical faults first is quite desirable. Moreover, since the purpose of the test case is given, the TCG users can realize the quality of the test and have certain confidence about the reliability of the IUT.

A prototype of a test case generator has been implemented using the TGF method. The system contains over 5,000 lines of C++, yacc, and lex programs. The program takes a specification and a fault model (both specified in EFSM) as input and produces test cases. A part of the system that resolves the truth value of the predicates are still absent. Currently, the program cannot determine the truth value if input values (such as $\nu\kappa$, $\nu^2\kappa$, etc.) are contained in the predicate. The program will ask the users if there are possible input values that can make the predicate evaluate false. If the user answers yes, a test case will be found. Otherwise, the program continues its search.

The proposed method is based on EFSM. Since Estelle has been adopted as an ISO standard specification language for EFSM, it is interesting to extend TGF to accept protocol specifications written in Estelle. The next chapter proposes an extension of TGF as a TCG method for Estelle.

CHAPTER VIII

TGF for Estelle

Since Estelle has been adopted as a standard specification language for communication protocols, it is interesting to extend TGF from EFSM to Estelle. In this chapter, an automatic test case generation method for Estelle is proposed. Axioms for Estelle statements are defined and Procedure **TGF** can be used to generate test cases for Estelle specifications with given fault models. The same method is also extended to generate test cases for multiple modules running concurrently. In addition, a heuristic search method is also presented to improve the performance of TGF.

8.1 Axioms for Estelle

Estelle specifications can be written in their *normal specification form*, in which every transition is specified in the following format:

from h
to t
when $p.m(x_1, x_2, ..., x_n)$
provided $\mathcal{B}$
begin S **end**;

where h and t, respectively, are the head and tail states of the transition, p is an interaction point, m is a message received from p with arguments x_1 through x_n, and $\mathcal{B}$ is a boolean expression indicating the condition under which a sequence of

Pascal statements, S, can be executed. A transition can be divided into two simpler transitions. As shown in Figure 57, the first transition contains **when** and **provided** clauses, and the second one executes S. An auxiliary state u is placed between the two transitions. Hence, the axiom for an Estelle transition can be defined as follows:

Axiom 8.1.1

$$\begin{aligned} \beta[h] = \; & \{\nu c_p = m(\nu x_1, \nu x_2, ..., \nu x_n)\} \cap \\ & \{(\nu x_1 = \nu\kappa) \wedge (\nu x_2 = \nu^2\kappa) \wedge \cdots \wedge (\nu x_n = \nu^n\kappa)\} \cap \\ & \{\nu\tau = \nu c_p\} \cap \\ & \{\mathcal{B}\} \cap \\ & \beta[u]_{\nu c_p, \nu x_1, ..., \nu x_n, \nu^n\kappa, \nu\tau} \end{aligned} \tag{8.1}$$

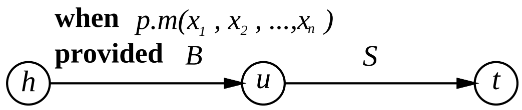


Figure 57: A transition can be divided into two simpler transitions.

There are five terms in the axiom. The first term shows what message is received from channel c_p. Here, c_p is treated as a variable, so that the operator ν can be used to distinguish the current input at channel c_p from the previous ones. Notation "$\nu c_p = m(\nu x_1, \nu x_2, ..., \nu x_n)$" means that the current data received from c_p (denoted νc_p) is message m with parameters x_1 through x_n. Since x_1 to x_n has been assigned new input values, they are upgraded and denoted by νx_1 to νx_n. The values of νx_1 to νx_n are shown in the second term, in which an auxiliary variable κ is used to denote different input values. Whenever an input value is needed, κ is upgraded once and assigned to the variable that needs the input, so that every input comes from a different version of κ and thus can be distinguished. The third term is used to arrange the external events in chronicle order. Similar to the imaginary variable κ, variable τ is upgraded every time when an external event happens. Axiom "$\nu\tau = \nu c_p$" denotes that the current external event is at channel c_p. The fourth term indicates that the condition in $\mathcal{B}$ must be satisfied. The last term means that the scenario set developed after the transition contains those scenarios developed after state u with updated variable versions. The versions are those that have been modified in the previous terms, which include νc_p, νx_1 to νx_n, $\nu^n\kappa$ (since κ has been upgraded n times), and $\nu\tau$.

The remaining problem is to define the axioms for S in order to obtain $\beta[u]$ in terms of $\beta[t]$. In the following sections, the notations A_s and V_s represent the assertion and the set of new versions developed after executing statement or statement sequence s, respectively. The axioms for Estelle statements are defined in Sections 8.1.1 to 8.1.7.

8.1.1 Empty Statements

For convenient, the axiom of an empty statement is defined as follows:

Axiom 8.1.2 *If S is empty,*

$$\beta[u] = \beta[t] \tag{8.2}$$

It is trivial that when there is no statement between u and t, the behavior of the two states is identical.

8.1.2 Sequential Statements

Axiom 8.1.3 *If S is "s; S'," where s is a statement and S' is a sequence of statements,*

$$\beta[u] = A_s \cap (A_{S'})_{V_s} \cap \beta[t]_{V_s \uplus V_{S'}} \tag{8.3}$$

Sequential statements can be treated as a serial composition of several single-statement transitions. Therefore, the axiom is similar to the equation for Figure 50b. The meaning of the axiom is that every scenario developed after state u must satisfy the restriction imposed by transition s, which is denoted by assertion A_s. If s had not been considered, executing S' along would result in assertion $A_{S'}$. Since s has been executed first, some variables used in s may have been upgraded. Therefore, $A_{S'}$ should upgrade those variables shown in V_s (denoted $(A_{S'})_{V_s}$). The variables upgraded by both s and S' are denoted by $V_s \uplus V_{S'}$. Had s and S' not been executed, the scenarios developed after state t would be $\beta[t]$. Since they are executed, the scenarios will be $\beta[t]_{V_s \uplus V_{S'}}$.

8.1.3 Assignment Statements

Axiom 8.1.4 *If S is "$x := e$," where x is a variable and e is an expression,*

$$\beta[u] = \{\nu x = e\} \cap \beta[t]_{\nu x} \tag{8.4}$$

The axiom means that the scenarios that can be developed after state h is the same as the scenarios developed after state t with one restriction that the new version of variable x is equal to the result of expression e (denoted by assertion $\{\nu x = e\}$).

Since variable x has been upgraded to one version higher than the original one, any development of variable x after state t should start from the upgraded version. Therefore, the second term of the axiom is denoted by $\beta[t]_{\nu x}$.

8.1.4 Output Statements

Axiom 8.1.5 *If S is "***output** *$p'.m'(e_1, e_2, ..., e_n)$," where p' is an interaction point, m' is an output message, and expressions e_1 to e_n are the arguments of m', then*

$$\beta[u] = \{\nu c_{p'} = m'(e_1, e_2, ..., e_n)\} \cap \{\nu\tau = \nu c_{p'}\} \cap \beta[t]_{\nu c_{p'}, \nu\tau} \tag{8.5}$$

where $c_{p'}$ is the corresponding channel of p'.

Similar to **when** clause, an output statement can be viewed as assigning the values in the output message to some auxiliary variables dedicated for recording outputs. In this way, an output statement will be identical to an assignment statement. Therefore, the channel corresponding to IP, p', is treated as a variable and $\nu c_{p'} = m'(e_1, e_2, ..., e_n)$ indicates that the next event at channel $c_{p'}$ is $m'(e_1, e_2, ..., e_n)$. The second term appends the event into the sequence of all external events recorded by τ. Since $c_{p'}$ and τ have been updated once in the first two terms, they should be upgraded in $\beta[t]$. Therefore, the third term is denoted by $\beta[t]_{\nu c_{p'}, \nu\tau}$.

8.1.5 Selection Statements

Axiom 8.1.6 *If S is "***if** *$\mathcal{B}$* **then** *S_1* **else** *S_2," where $\mathcal{B}$ is a boolean expression and S_1 and S_2 are two sequences of statements,*

$$\beta[u] = (\{\mathcal{B}\} \cap A_{S_1} \cap \beta[t]_{V_{S_1}}) \cup (\{\neg\mathcal{B}\} \cap A_{S_2} \cap \beta[t]_{V_{S_2}}) \tag{8.6}$$

A selection statement can be viewed as a parallel composition of two transitions. One can be executed only when condition $\mathcal{B}$ is satisfied, and the other can be executed only when it is not. Therefore, the result is the union of both.

8.1.6 Functions and Procedures

A procedure or function call can be decomposed into three steps. First, the arguments are assigned to the parameters. Next, the procedure or function body is executed. Finally, the results are returned to the caller. Therefore, the axioms for procedures and functions are defined according to these steps.

Axiom 8.1.7 *Let F be a function defined as follows:*

Function $F(x_1, x_2, ..., x_n)$;
  begin
  S_F;
  $F := e$
  end;

where $x_1, x_2, ..., x_3$ are parameters of F, S_F is a sequence of statements, and e is an expression. If S is "$y := F(a_1, a_2, ..., a_n)$," where $a_1, a_2, ..., a_n$ are arguments, then

$$\begin{aligned} \beta[u] \;\; = \;\; & \{(\nu x_1 = a_1 \wedge \nu x_2 = a_2 \wedge ... \wedge \nu x_n = a_n)\} \cap \\ & (A_{S_F})_{V'} \cap \\ & \{\nu y = e\}_{V_{S_F} \uplus V'} \cap \\ & \beta[t]_{V_{S_F} \uplus V' \uplus \{\nu y\}} \end{aligned} \tag{8.7}$$

where $V' = \{\nu x_1, \nu x_2, ..., \nu x_n\}$.

The first term of the axiom shows that the arguments (the a's) are assigned to the parameters (the x's). Since the parameters receive new values, their versions are upgraded and included in V'. The assertion imposed by the body of the function (A_{S_F}) also restricts the development of the scenarios after the function call. Therefore, it is included in the second term. In addition, since the values of parameters have been changed, those parameters referenced by A_{S_F} must be upgraded as well. Thus, the second term is denoted by $(A_{S_F})_{V'}$. The third term shows that after the function call, variable y receives the value from expression e (thus $\{\nu y = e\}$). In addition, any variable in V_{S_F} or V' needs to be upgraded accordingly (thus $\{\nu y = e\}_{V_{S_F} \uplus V'}$). Finally, the new versions generated by the above are contained in $V_{S_F} \uplus V' \uplus \nu y$, so the variables in $\beta[t]$ need to be upgraded accordingly.

Axiom 8.1.8 *Let P be a procedure defined as follows:*

> **Procedure** $P(x_1,\ x_2,\ ...,\ x_n,$ **var** $y_1,$ **var** $y_2,\ ...,$ **var** $y_m)$;
> **begin**
> S_P
> **end;**

where $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_m$ are variables and S_P is a sequence of statements. If S is "$P(a_1, a_2, ..., a_n, b_1, b_2, ..., b_m)$," where $a_1, a_2, ..., a_n$ and $b_1, b_2, ..., b_m$ are arguments, then

$$\begin{aligned}\beta[u] \quad = \quad & \{(\nu x_1 = a_1 \wedge \nu x_2 = a_2 \wedge ... \wedge \nu x_n = a_n)\wedge \\ & (\nu y_1 = b_1 \wedge \nu y_2 = b_2 \wedge ... \wedge \nu y_m = b_m)\}\cap \\ & (A_{S_P})_{V'}\cap \\ & \{\nu b_1 = y_1 \wedge \nu b_2 = y_2 \wedge ... \wedge \nu b_n = y_n\}_{V_{S_P} \uplus V'}\cap \\ & \beta[t]_{V_{S_P} \uplus V' \uplus V''}\end{aligned} \tag{8.8}$$

where $V' = \{\nu x_1, \nu x_2, ..., \nu x_n, \nu y_1, \nu y_2, ..., \nu y_m\}$ and $V'' = \{\nu b_1, \nu b_2, ..., \nu b_m\}$.

Similar to a function call, a procedure call assigns arguments to parameters first (the first term). Then, the same condition that S_P imposes is considered in the second term. After a procedure is finished, those pass-by-reference variables must receive their new values (the third term). Finally, the procedure call is finished and the scenarios must follow whatever restriction imposed by the behavior of state t

(the last term). Note that arguments a's can be expressions, while b's can only be variables.

It is assumed that every variable has its own name. Although this is not true in a real specification, name conflicts can always be resolved by appending the corresponding function or procedure names to the variable names.

8.1.7 Iterations

Defining axioms for iterations is a difficult task. This is because other types of statements have fixed control flow, so that their axioms can be decided before the statements are executed. On the contrary, the control flow of an iteration statement is decided by the current value of the variables. For example, how many times the statement "**for** $i := 1$ **to** n **do** S;" executes S cannot be known until the value of n is decided.

To deal with iterations, *loop invariant* has been used to specify the conditions that never change in the iterations. However, constructing loop invariant requires human intuition and has been the most criticized part in automatic program proving. Therefore, instead of using loop invariant, a straight forward approach is used. Before describing our approach, the following operation is defined first.

Definition 8.1.9 *Let V be a version set. Then,*

$$\begin{aligned} V^n &= \underbrace{V \uplus V \uplus \ldots \uplus V}_{n} \\ &= \{\nu^{nk}w \mid \nu^k w \in V\} \end{aligned} \tag{8.9}$$

In other words, V^n indicates that the variables in V are upgraded n times. Now we are ready to deal with iterations as follows.

Axiom 8.1.10 *If S is "***while** *$\mathcal{B}$* **do** *S'," where $\mathcal{B}$ is a boolean expression and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=0}^{n-1} [\{\mathcal{B}\} \cap A_{S'}]_{V_{S'}^k} \cap [\{\neg\mathcal{B}\} \cap \beta[t]]_{V_{S'}^n} \tag{8.10}$$

The **while** statement can be decomposed into the state-transition diagram shown in Figure 58a. In the figure, when $\mathcal{B}$ is satisfied, the EFSM moves to state v, executes the statement S' and returns to state u. If $\mathcal{B}$ is still satisfied, S' is executed again. Otherwise, the EFSM reaches t and finishes. Let $\beta[u]_i$ denote the i-th time state u is visited after S' is executed. Assuming that S' has been executed n times, then

$$\begin{aligned}
\beta[u] &= (\{\mathcal{B}\} \cap A_{S'}) \cap (\beta[u]_1)_{V_{S'}} \\
&= (\{\mathcal{B}\} \cap A_{S'}) \cap (\{\mathcal{B}\} \cap A_{S'})_{V_{S'}} \cap (\beta[u]_2)_{V_{S'}^2} \\
&\vdots \\
&= (\{\mathcal{B}\} \cap A_{S'}) \cap (\{\mathcal{B}\} \cap A_{S'})_{V_{S'}} \cap \cdots \cap (\{\mathcal{B}\} \cap A_{S'})_{V_{S'}^{n-1}} \cap (\beta[u]_n)_{V_{S'}^n} \\
&= \textstyle\bigcap_{k=0}^{n-1} (\{\mathcal{B}\} \cap A_{S'})_{V_{S'}^k} \cap (\beta[u]_n)_{V_{S'}^n} \\
&= \textstyle\bigcap_{k=0}^{n-1} (\{\mathcal{B}\} \cap A_{S'})_{V_{S'}^k} \cap (\{\neg\mathcal{B}\} \cap \beta[t]_{V_{S'}^n})
\end{aligned} \tag{8.11}$$

It is possible to construct a data structure to represent "$\bigcap_{k=0}^{n-1}(\{\mathcal{B}\} \cap A_{S'})_{V_{S'}^k}$" such that n remains variable. Later, when the behavior of the EFSM is analyzed to find a test case, $\beta[u]$ will be *expanded* and the actual value of n will be determined then.

The **repeat** and **for** statements are similar to the **while** statement. Figures 58b to 58d shows the interpretation of the **repeat** and **for** statements in state-transition diagrams and their axioms are defined in Axioms 8.1.11 to 8.1.13. Note that the actual value of n and m can be determined later when necessary.

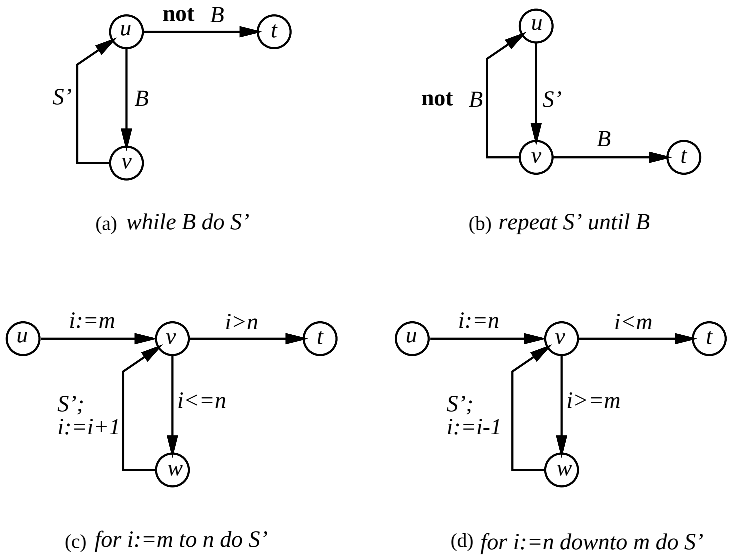


Figure 58: State-transition diagrams for iteration statements: (a) **while** $\mathcal{B}$ **do** S', (b) **do** S' **until** B, (c) **for** $i := m$ **to** n **do** S', and (d) **for** $i := m$ **downto** n **do** S'

Axiom 8.1.11 *If S is "***repeat** *S'* **until** *$\mathcal{B}$," where $\mathcal{B}$ is a boolean expression and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=1}^{n} [(A_{S'})_{V_{S'}^{k-1}} \cap \{\neg\mathcal{B}\}_{V_{S'}^{k}}] \cap [\{\mathcal{B}\} \cap \beta[t]]_{V_{S'}^{n}} \tag{8.12}$$

Axiom 8.1.12 *If S is "***for** *$i := m$* **to** *n* **do** *S'," where i is an integer, m and n are integer expressions, and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=0}^{n-m} [\{\nu i = k + m\} \cap A_{S'}]_{(\nu i \uplus V_{S'})^{k}} \cap \beta[t]_{V_{S'}^{n-m+1}} \tag{8.13}$$

Axiom 8.1.13 *If S is "**for** $i := n$ **downto** m **do** S'," where i is an integer, m and n are integer expressions, and S' is a sequence of statements, then*

$$\beta[u] = \bigcap_{k=0}^{n-m} [\{\nu i = n - k\} \cap A_{S'}]_{(\nu i \uplus V_{S'})^k} \cap \beta[t]_{V_{S'}^{n-m+1}} \tag{8.14}$$

8.1.8 An Example

Having the axioms defined in the previous section, the behavior of the EFSM in Figure 6 can be derived as follows. For simplification, the initial transition is ignored. According to Axiom 8.1.1,

$$\beta[S_1] = \{\nu c_u = send(\nu d)\} \cap \{\nu d = \nu \kappa\} \cap \{\nu \tau = \nu c_u\} \cap \{\mathbf{true}\} \cap \beta[a_{T_1}]_{\nu d, \nu c_u, \nu \kappa, \nu \tau} \tag{8.15}$$

where a_{T_1} is an auxiliary state for transition T_1. According to Axiom 8.1.5,

$$\beta[a_{T_1}] = \{\nu c_r = msg(d, b)\} \cap \{\nu \tau = \nu c_r\} \cap \beta[S_2]_{\nu c_r, \nu \tau} \tag{8.16}$$

Therefore,

$$\beta[a_{T_1}]_{\nu d, \nu c_u, \nu \kappa, \nu \tau} = \{\nu c_r = msg(\nu d, b)\} \cap \{\nu^2 \tau = \nu c_r\} \cap \beta[S_2]_{\nu c_r, \nu^2 \tau, \nu d, \nu c_u, \nu \kappa} \tag{8.17}$$

Substituting the right-hand side of the above equation into $\beta[S_1]$, the behavior of the ABP at state S_1 is derived as the one in Figure 59. Similarly, $\beta[S_2]$ can be derived from Axioms 8.1.1, 8.1.4, 8.1.5, and 8.1.6. The first half of $\beta[S_2]$ in Figure 59 is derived from the statements in T_2 and the second from statements in T_3. Since both T_2 and T_3 are outgoing transitions of S_2, their assertions are composed in parallel to form the behavior of the ABP at state S_2.

With the behavior of the ABP in hand, the formulas that represent the behavior can be *expanded*. Expanding the formulas is like symbolically executing the EFSM to

$$
\begin{aligned}
\beta[S_1] &= \{(\nu c_u = send(\nu d)) \wedge (\nu d = \nu \kappa) \wedge (\nu \tau = \nu c_u)\} \cap \\
&\quad \{(\nu c_r = msg(\nu d, b)) \wedge (\nu^2 \tau = \nu c_r)\} \cap \\
&\quad \beta[S_2]_{\nu c_u, \nu c_r, \nu d, \nu \kappa, \nu^2 \tau} \\
\beta[S_2] &= (\{(\nu c_r = ack(\nu a)) \wedge (\nu a = \nu \kappa) \wedge (\nu \tau = \nu c_r)\} \cup \\
&\quad \{(\nu a \neq b) \wedge (\nu^2 c_r = msg(d, b)) \wedge (\nu^2 \tau = \nu^2 c_r)\} \cap \\
&\quad \beta[S_2]_{\nu a, \nu^2 c_r, \nu \kappa, \nu^2 \tau}) \cup \\
&\quad (\{(\nu c_r = ack(\nu a)) \wedge (\nu a = \nu \kappa) \wedge (\nu \tau = \nu c_r)\} \cap \\
&\quad \{(\nu a = b) \wedge ((b = 0 \wedge \nu b = 1) \vee (b = 1 \wedge \nu b = 0))\} \cap \\
&\quad \beta[S_1]_{\nu a, \nu b, \nu c_r, \nu \kappa, \nu \tau})
\end{aligned}
$$

Figure 59: The behavior function of the ABP in Figure 22

see what scenarios can be developed after each state. For example, $\beta[S_1]$ of the ABP can be expanded as follows:

$$
\begin{aligned}
\beta[S_1] &= \{(\nu c_u = send(\nu d)) \wedge (\nu d = \nu \kappa) \wedge (\nu \tau = \nu c_u) \wedge (\nu c_r = msg(\nu d, b)) \wedge (\nu^2 \tau = \nu c_r)\} \cap \\
&\quad \beta[S_2]_{\nu c_u, \nu c_r, \nu d, \nu \kappa, \nu^2 \tau} \\
&= \{(\nu c_u = send(\nu d)) \wedge (\nu d = \nu \kappa) \wedge (\nu \tau = \nu c_u) \wedge (\nu c_r = msg(\nu d, b)) \wedge (\nu^2 \tau = \nu c_r)\} \cap \\
&\quad (\{(\nu^2 c_r = ack(\nu a)) \wedge (\nu a = \nu^2 \kappa) \wedge (\nu^3 \tau = \nu^2 c_r) \wedge \\
&\quad (\nu a \neq b) \wedge (\nu^3 c_r = msg(\nu d, b)) \wedge (\nu^4 \tau = \nu^3 c_r)\} \cap \\
&\quad \beta[S_2]_{\nu c_u, \nu d, \nu a, \nu^3 c_r, \nu^2 \kappa, \nu^4 \tau}) \cup \\
&\quad (\{(\nu^2 c_r = ack(\nu a)) \wedge (\nu a = \nu^2 \kappa) \wedge (\nu^3 \tau = \nu^2 c_r) \wedge (\nu a = b) \wedge \\
&\quad ((b = 0 \wedge \nu b = 1) \vee (b = 1 \wedge \nu b = 0))\} \cap \\
&\quad \beta[S_1]_{\nu c_u, \nu d, \nu a, \nu b, \nu^2 c_r, \nu^2 \kappa, \nu^3 \tau}) \\
&= \cdots
\end{aligned}
$$

The second formula is derived by substituting $\beta[S_2]$ into the first formula.

8.2 Test Case Generation

8.2.1 Test Case Generation for Single-Module Specifications

Figure 13 shows how test cases of the mutant in Figure 60 can be generated using Procedure **TGF** in Figure 56. For convenience, the j-th item from the top of the i-th